
RedisVL

Release 0.27.2

Redis Applied AI

Oct 08, 2026

CONTENTS

1	Explore the Docs	3
1.1	Concepts	3
1.2	Guides	45
1.3	Example Gallery	261
1.4	RedisVL API	263
	Index	449

```
pip install redisvl
```

```
docker run -d --name redis -p 6379:6379 redis:8.4
```

Or connect to [Redis Cloud](#) for a managed experience.

→ [Getting Started](#)

EXPLORE THE DOCS

- Concepts Understand how RedisVL works. Architecture, search fundamentals, and extension patterns.
- User Guides Step-by-step tutorials. Installation, getting started, and deep dives on every feature.
- Examples Real-world applications. RAG pipelines, chatbots, recommendation systems, and more.
- API Reference Complete API documentation. Classes, methods, parameters, and examples.

1.1 Concepts

Foundational knowledge for building AI applications with RedisVL. These concepts are language-agnostic and apply across all RedisVL implementations.

- Architecture How RedisVL components connect: schemas, indexes, queries, and extensions.
 - Search & Indexing Schemas, fields, documents, storage types, and query patterns.
 - Index Migrations How RedisVL handles migration planning, rebuilds, and future shadow migration.
 - Field Attributes Configure sortable, no_index, index_missing, and other field options.
 - Query Types Vector, filter, text, hybrid, and multi-vector query options.
 - Utilities Vectorizers for embeddings and rerankers for result optimization.
 - MCP How RedisVL exposes an existing Redis index to MCP clients through a stable tool contract.
 - Extensions Pre-built patterns: caching, message history, and semantic routing.

1.1.1 Architecture

RedisVL sits between your application and Redis, providing a structured way to define, populate, and query vector search indexes.

The Core Pattern

Every RedisVL application follows a consistent workflow: **define** → **create** → **load** → **query**.

First, you define a **schema** that describes your data. The schema specifies which fields exist, what types they are, and how they should be indexed. This includes declaring vector fields with their dimensionality and the algorithm Redis should use for similarity search.

Next, you create an **index** in Redis based on that schema. The index is a persistent structure that Redis uses to make searches fast. Creating an index tells Redis how to organize and access your data.

Then you **load** your data. Documents are stored as Redis Hash or JSON objects. As documents are written, Redis automatically indexes them according to your schema—no separate indexing step required.

Finally, you **query** the index. RedisVL provides query builders that construct Redis search commands for you. You can search by vector similarity, filter by metadata, combine multiple criteria, or mix full-text search with semantic search.

This pattern applies whether you're building a simple semantic search or a complex multi-modal retrieval system.

Schemas as Contracts

The schema is the source of truth for your index. It defines the contract between your data and Redis.

A schema includes the index name, a key prefix (so Redis knows which keys belong to this index), the storage type (Hash or JSON), and a list of field definitions. Each field has a name, a type, and type-specific configuration.

For vector fields, you specify the dimensionality (which must match your embedding model's output), the distance metric (cosine, Euclidean, or inner product), and the indexing algorithm. These choices are locked in when the index is created—changing them requires building a new index.

The schema can be defined programmatically in Python or loaded from a YAML file. YAML schemas are useful for version control, sharing between environments, and keeping configuration separate from code.

Query Composition

RedisVL's query builders let you compose search operations without writing raw Redis commands.

Vector queries find the K most similar items to a query vector. You provide an embedding, and Redis returns the nearest neighbors according to your configured distance metric.

Range queries find all vectors within a distance threshold. Instead of asking for the top K, you're asking for everything "close enough" to a query point.

Filter queries narrow results by metadata. You can filter on text fields, tags, numeric ranges, and geographic areas. Filters apply before the vector search, reducing the candidate set.

Hybrid queries combine keyword search with semantic search. This is useful when you want to match on specific terms while also considering semantic relevance.

These query types can be combined. A typical pattern is vector search with metadata filters—for example, finding similar products but only in a specific category or price range.

Extensions as Patterns

Extensions are higher-level abstractions built on RedisVL’s core primitives. Each extension encapsulates a common AI workflow pattern.

Semantic caching stores LLM responses and retrieves them when similar prompts are seen again. This reduces API costs and latency without requiring exact-match caching.

Message history stores conversation turns and retrieves context for LLM prompts. It can retrieve by recency (most recent messages) or by relevance (semantically similar messages).

Semantic routing classifies queries into predefined categories based on similarity to reference phrases. This enables intent detection, topic routing, and guardrails.

Each extension manages its own Redis index internally. You interact with a clean, purpose-specific API rather than managing schemas and queries yourself.

Related concepts: *Search & Indexing* covers schemas and field types in detail. *Query Types* explains the different query types available.

Learn more: *Getting Started* covers the core workflow. *Extensions* explains each extension pattern in detail.

1.1.2 Search & Indexing

Vector search in Redis works differently from traditional databases. Understanding the underlying model helps you design better schemas and write more effective queries.

How Redis Indexes Work

Redis search indexes are secondary structures that sit alongside your data. When you create an index, you’re telling Redis: “Watch all keys that match this prefix, and build a searchable structure from these specific fields.”

The index doesn’t store your data—it references it. Your documents live as Redis Hash or JSON objects, and the index maintains pointers and optimized structures for fast lookup. When you write a document, Redis automatically updates the index. When you delete a document, the index entry is removed.

This design means searches are fast (the index is optimized for queries) while writes remain efficient (only the affected index entries are updated). It also means you can have multiple indexes over the same data with different field configurations.

Field Types and Their Purpose

Each field type serves a different search use case.

Text fields enable full-text search. Redis tokenizes the content, applies stemming (so “running” matches “run”), and builds an inverted index. Text search finds documents containing specific words or phrases, ranked by relevance.

Tag fields are for exact-match filtering. Unlike text fields, tags are not tokenized or stemmed. A tag value is treated as an atomic unit. This is ideal for categories, statuses, IDs, and other discrete values where you want exact matches.

Numeric fields support range queries and sorting. You can filter for values greater than, less than, or between bounds. Numeric fields are also used for sorting results.

Geo fields enable location-based queries. You can find documents within a radius of a point or within a bounding box.

Vector fields enable similarity search. Each document has an embedding vector, and queries find documents whose vectors are closest to a query vector. This is the foundation of semantic search.

Vector Indexing Algorithms

Vector similarity search requires specialized data structures. Redis offers three algorithms, each with different trade-offs.

Flat indexing performs exact nearest-neighbor search by comparing the query vector against every indexed vector. This guarantees finding the true closest matches, but search time grows linearly with dataset size. Use flat indexing for small datasets (under ~100K vectors) where exact results matter.

HNSW (Hierarchical Navigable Small World) is an approximate algorithm that builds a multi-layer graph structure. Queries navigate this graph to find approximate nearest neighbors in logarithmic time. HNSW typically achieves 95-99% recall (meaning it finds 95-99% of the true nearest neighbors) while being orders of magnitude faster than flat search on large datasets. This is the default choice for most applications.

SVS (Scalable Vector Search) is designed for very large datasets with memory constraints. It supports vector compression techniques that reduce memory footprint at the cost of some recall. SVS is useful when you have millions of vectors and memory is a limiting factor.

The algorithm choice is made at index creation and cannot be changed without rebuilding the index.

Distance Metrics

When comparing vectors, you need a way to measure how “close” two vectors are. Redis supports three distance metrics.

Cosine distance measures the angle between vectors, ignoring their magnitude. Two vectors pointing in the same direction have distance 0; opposite directions have distance 2. Cosine is widely used because most embedding models produce vectors where direction encodes meaning and magnitude is less important. Similarity equals 1 minus distance.

Euclidean distance (L2) measures the straight-line distance between vector endpoints. Unlike cosine, it considers magnitude. Euclidean distance ranges from 0 to infinity.

Inner product (IP) is the dot product of two vectors. It combines both direction and magnitude. When vectors are normalized (magnitude 1), inner product equals cosine similarity. Inner product can be negative and ranges from negative infinity to positive infinity.

Choose your metric based on how your embedding model was trained. Most text embedding models use cosine.

Storage: Hash vs JSON

Redis offers two storage formats for documents.

Hash storage is a flat key-value structure where each field is a top-level key. It’s simple, fast, and works well when your documents don’t have nested structures. Field names in your schema map directly to hash field names.

JSON storage supports nested documents. You can store complex objects and use JSONPath expressions to index nested fields. This is useful when your data is naturally hierarchical or when you want to store the original document structure without flattening it.

The choice affects how you structure data and how you reference fields in schemas. Hash is simpler; JSON is more flexible.

Index Lifecycle

Indexes have a straightforward lifecycle: create, use, and eventually delete.

Creating an index registers the schema with Redis. The `create()` method sends the schema definition to Redis, which builds the necessary data structures. If an index with the same name exists, you can choose to overwrite it (optionally dropping existing data) or raise an error.

Checking existence with `exists()` tells you whether an index is registered in Redis. This is useful before creating (to avoid errors) or before querying (to ensure the index is ready).

Connecting to existing indexes is possible with `from_existing()`. If an index was created elsewhere—by another application, a previous deployment, or the CLI—you can connect to it by name. RedisVL fetches the index metadata from Redis and reconstructs the schema automatically.

```
# Connect to an index that already exists in Redis
index = SearchIndex.from_existing("my-index", redis_url="redis://localhost:6379")
```

Deleting an index with `delete()` removes the index definition from Redis. By default, this also deletes all documents associated with the index. Pass `drop=False` to keep the documents while removing only the index structure.

Clearing data with `clear()` removes all documents from the index without deleting the index itself. The schema remains intact, ready for new data.

Search Visibility After Writes

On Redis 8.8 and later, deployments with a nonzero `search-workers` setting use background workers for search indexing. Under concurrent write and query load, a document can already exist in the Redis keyspace but not yet be visible to a search issued immediately after the write. For example, a query directly after `load()` can temporarily return fewer documents than were loaded:

```
index.load(records)
results = index.query(query) # may not include every newly loaded document yet
```

This is a short indexing delay, not data loss. A later query sees the documents after the index catches up. The measurements reported in [Issue #662](#) typically converged within 1–3 ms, but that observation is not a latency guarantee and the window can vary with workload and server capacity.

Applications that require strict search read-after-write behavior can start Redis with `--search-workers 0`. This is a Redis server setting, not a RedisVL option: it keeps indexing in the foreground but gives up the throughput benefits of multithreaded background indexing.

Bulk Delete and Update

Redis has no server-side “delete/update by query”, so mutating many documents traditionally means scanning for keys and issuing writes yourself. RedisVL wraps that pattern behind two filter-driven methods (available on both `SearchIndex` and `AsyncSearchIndex`).

Deleting by filter resolves every document matching a filter expression and removes it with non-blocking `UNLINK`, in batches:

```
from redisvl.query.filter import Tag, Num

# Remove every archived document from before 2020
index.drop_by_filter((Tag("status") == "archived") & (Num("year") < 2020))
```

Updating by filter applies a partial field update to every match. Fields you don't mention are left untouched—hash fields are written with HSET, and JSON documents are merged at the root with JSON.MERGE (RFC 7396: nested objects merge recursively, arrays are replaced wholesale, and a None value deletes that path):

```
# Mark all draft documents as published, leaving other fields intact
index.update_by_filter(Tag("status") == "draft", {"status": "published"})
```

Two caveats for `update_by_filter` values:

- **No schema validation.** Unlike `load()`, values are written as-is—pre-encode vectors/bytes and format numerics as your schema expects, or you may corrupt query results.
- **JSON keys must match the document layout, not the schema field name.** Values merge at the document root (`$`). If a field is indexed at a nested path (e.g. `$.metadata.status`), pass the correspondingly nested mapping (`{"metadata": {"status": "published"}}`); passing the flat field name writes the wrong path and leaves the indexed field unchanged. Only static values are supported (no callable/expression transforms).

Both methods share the same safety-oriented options and return a `BulkResult`:

- The return value carries `matched` (documents matching the filter), `processed` (documents actually affected), `completed` (False if a delete stopped early at its runaway backstop; always True for update), and `dry_run`. Read the fields explicitly—`result.processed`, `result.completed`, etc.
- `dry_run=True` reports how many documents *would* be affected—via a count query—without changing anything.
- A match-all filter (empty, `"*"`, or `None`) is refused unless you pass `allow_all=True`; use `clear()` when you intentionally want to empty the index. This guard catches the canonical match-all forms only—it is a convenience backstop, not a security control.
- `on_progress(processed, matched)` is called after each write batch for observability on large operations (it runs synchronously—don't pass a coroutine—and raising from it aborts the run). Note `update_by_filter` resolves all matching keys before it writes, so progress callbacks begin only once the write phase starts. `batch_size` tunes how many documents are processed per round-trip.

The related `drop_documents()` and `drop_keys()` helpers delete by document ID or full Redis key and batch large inputs. On Redis Cluster, `drop_keys()` unlinks per-key so it works across hash slots, while `drop_documents()` requires the target keys to share a hash tag and raises `ValueError` otherwise.

Durability and partial failure. Because Redis has no server-side delete/update-by-query, these operations run as a series of batched writes rather than a single transaction—they are **not atomic across the match set** and there is no rollback. The unit of atomicity is a single document: each key is deleted with one UNLINK, and each document is updated with one HSET or JSON.MERGE, so you never get a half-deleted key or a half-updated document. But batches are applied incrementally, so a crash or connection error mid-run leaves some documents changed and the rest untouched.

The intended recovery is simply to **re-run the same call**: both operations are idempotent, so a repeat pass removes (or re-sets) only what still needs it and converges on the desired state. There is no built-in checkpoint or resume token—`on_progress` reports live progress but is not a restart point. `update_by_filter` resolves keys before it writes, so a document can be deleted by another client in between; each write is applied only if the key still exists (atomically), so such a document is **skipped, not recreated** as a partial document—it just isn't counted in `processed`, which is why `processed` can be less than `matched` under concurrent deletion. Conversely, `processed` never double-counts: a document is written *at most* once, so `processed` counts *distinct* documents even though the cursor that resolves them can return the same document on more than one page. It can exceed `matched` only if the match set grew after the initial count—a concurrent insert, or a count query that itself returned a partial total under a server-side search timeout. Note there is no matching *at least* once guarantee: a cursor may return fewer rows than `matched` (a **partial result** even when it reports a cursor id of 0, or rows missing during a slot migration on Redis Software), and a document the cursor never returns is never written—so `processed` short of `matched` is not by itself proof of a concurrent deletion. Note this guard is *existence*-based, not *identity*-based: if a key is deleted and a different document is recreated at the

same key mid-run, the update can land on that new document. Running against a quiescent partition (below) avoids all of these concurrency cases.

Operational considerations at scale. These are single-threaded, foreground operations that issue one round-trip per batch; a very large match set is a long-running job. Keep the following in mind for production use:

- **Live-traffic impact.** Every delete/update also updates the search index synchronously. Running a large bulk job against a node that is serving queries competes for CPU and reindex bandwidth and can raise query latency—prefer off-peak windows, or narrow the filter and run in waves.
- **Memory.** `drop_by_filter` holds only one batch of keys at a time ($O(\text{batch_size})$). `update_by_filter` must resolve *all* matching keys before it can write (an open aggregation cursor can't be read while the index is being written), so its client memory grows with the match count—roughly the total size of the matched keys, plus about a third again for the set that de-duplicates them (an aggregation cursor can return the same document on more than one page, so keys are de-duplicated to guarantee one write per document). Budget on the order of 120 MB per million 40-character keys. For very large match sets, partition the filter (see below) rather than updating everything in one call.
- **Redis Cluster.** Cross-slot multi-key commands aren't allowed, so writes are issued per key (no pipelining across slots)—expect this to be slower on Cluster for large match sets.
- **Resumability.** There is no checkpoint; recovery is a full re-run. For very large corpora, **partition the filter** (e.g. by a tag or numeric range) and process partition-by-partition. This is the recommended pattern at scale: it bounds memory and per-run time, keeps each call independently retryable, and lets you spread load across off-peak windows.

Data Validation

RedisVL can validate data against your schema before loading it to Redis. This catches type mismatches, missing required fields, and invalid values early—before they cause problems in production.

Enable validation by setting `validate_on_load=True` when creating the index:

```
index = SearchIndex.from_dict(schema, redis_url="redis://localhost:6379", validate_on_
↪load=True)
```

When validation is enabled, each object is checked against the schema during `load()`. If validation fails, a `SchemaValidationError` is raised with details about which field failed and why.

Validation adds overhead, so it's typically enabled during development and testing, then disabled in production once you're confident in your data pipeline.

Schema Evolution

Redis doesn't support modifying an existing index schema. Once an index is created, its field definitions are fixed.

To change a schema, you create a new index with the updated configuration, reindex your data into it, update your application to use the new index, and then delete the old index. This pattern—create new, migrate, switch, drop old—is the standard approach for schema changes in production.

Planning your schema carefully upfront reduces the need for migrations, but the capability exists when requirements evolve.

RedisVL now includes a dedicated migration workflow for this lifecycle:

- `drop_recreate` for document-preserving rebuilds, including vector quantization (`float32` → `float16`)

That means schema evolution is no longer only a manual operational pattern. It is also a product surface in RedisVL with a planner, CLI, and validation artifacts.

Related concepts: *Field Attributes* explains how to configure field options like `sortable` and `index_missing`. *Query Types* covers the different query types available. *Index Migrations* explains migration modes, supported changes, and architecture.

Learn more: *Getting Started* walks through building your first index. *Choose a Storage Type* compares storage options in depth. *Query and Filter Data* covers query composition. *Migrate an Index* shows how to use the migration CLI in practice.

1.1.3 Index Migrations

Warning

The index migrator is an **experimental** feature. APIs, CLI commands, and on-disk formats (plans, checkpoints, backups) may change in future releases. Review migration plans carefully before applying to production indexes.

Redis Search indexes are immutable. To change an index schema, you must drop the existing index and create a new one. RedisVL provides a migration workflow that automates this process while preserving your data.

This page explains how migrations work and which changes are supported. For step by step instructions, see the *migration guide*.

Supported and blocked changes

The migrator classifies schema changes into two categories:

Change	Status
Add or remove a field	Supported
Rename a field	Supported
Change field options (sortable, separator)	Supported
Change key prefix	Supported
Rename the index	Supported
Change vector algorithm (FLAT, HNSW, SVS-VAMANA)	Supported
Change distance metric (COSINE, L2, IP)	Supported
Tune algorithm parameters (M, EF_CONSTRUCTION)	Supported
Quantize vectors (float32 to float16/bfloat16/int8/uint8)	Supported
Change vector dimensions	Blocked
Change storage type (hash to JSON)	Blocked
Add a new vector field	Blocked

Note: INT8 and UINT8 vector datatypes require Redis 8.0+. SVS-VAMANA algorithm requires Redis 8.2+ and Intel AVX-512 hardware.

Supported changes can be applied automatically using `rvl migrate`. The migrator handles the index rebuild and any necessary data transformations.

Blocked changes require manual intervention because they involve incompatible data formats or missing data. The migrator will reject these changes and explain why.

How the migrator works

The migrator uses a plan first workflow:

1. **Plan:** Capture the current schema, classify your changes, and generate a migration plan
2. **Review:** Inspect the plan before making any changes
3. **Apply:** Drop the index, transform data if needed, and recreate with the new schema
4. **Validate:** Verify the result matches expectations

This separation ensures you always know what will happen before any changes are made.

Migration mode: `drop_recreate`

The `drop_recreate` mode rebuilds the index in place while preserving your documents.

The process:

1. Drop only the index structure (documents remain in Redis)
2. For datatype changes, re-encode vectors to the target precision
3. Recreate the index with the new schema
4. Wait for Redis to re-index the existing documents
5. Validate the result

Tradeoff: The index is unavailable during the rebuild. Review the migration plan carefully before applying.

Index only vs document dependent changes

Schema changes fall into two categories based on whether they require modifying stored data.

Index only changes affect how Redis Search indexes data, not the data itself:

- Algorithm changes: The stored vector bytes are identical. Only the index structure differs.
- Distance metric changes: Same vectors, different similarity calculation.
- Adding or removing fields: The documents already contain the data. The index just starts or stops indexing it.

These changes complete quickly because they only require rebuilding the index.

Document dependent changes require modifying the stored data:

- Datatype changes (float32 to float16): Stored vector bytes must be re-encoded.
- Field renames: Stored field names must be updated in every document.
- Dimension changes: Vectors must be re-embedded with a different model.

The migrator handles datatype changes and field renames automatically. Dimension changes are blocked because they require re-embedding with a different model (application level logic).

Vector quantization

Changing vector precision from float32 to float16 reduces memory usage at the cost of slight precision loss. The migrator handles this automatically by:

1. Reading all vectors from Redis
2. Converting to the target precision
3. Writing updated vectors back
4. Recreating the index with the new schema

Typical reductions:

Metric	Value
Index size reduction	~50%
Memory reduction	~35%

Quantization time is proportional to document count. Plan for downtime accordingly.

Vector backups (mandatory for quantization)

Quantization mutates the raw bytes of every vector in place. If the migration is interrupted partway through, or if the converted bytes turn out to be unacceptable for your application, there is no way to recover the original precision from the quantized values. To make these migrations safe to run, the migrator **always writes a vector backup before mutating any data** when a quantization step is needed.

There is no opt-out. The previous `--keep-backup` flag and any code path that allowed quantizing without a backup have been removed.

Where backups are written

Pass `--backup-dir <dir>` (CLI) or `backup_dir="<dir>"` (Python API) to choose the location. If you do not supply one, or if you pass an empty string, the migrator raises a `ValueError` before any data is touched. This argument is required for every migration apply. Quantization migrations write `.header` and `.data` backup files there; multi-worker quantization also writes a `.manifest` file that lets the executor resume from worker shards after the source index has been dropped. Index-only migrations record the resolved directory in the report but do not write vector backup files.

Each hash index that mutates vector bytes produces backup files like:

```
<backup-dir>/
  migration_backup_<index_name>.header  # JSON: phase, progress counters, field metadata
  migration_backup_<index_name>.data    # Binary: length-prefixed batches of original
↪ vectors
  migration_backup_<index_name>.manifest # JSON: multi-worker shard resume metadata,
↪ when workers > 1
```

The migration report records the resolved `backup_dir` and any backup file prefixes used for the run. For index-only migrations and JSON datatype changes, the directory is still validated and recorded, but no vector backup files are written. Batch checkpoint state also records `backup_dir` so batch-resume can verify it is using the same recovery location.

Disk usage is roughly `num_docs × dims × bytes_per_element`. For 1M documents with 768-dimensional float32 vectors that is approximately 2.9 GB.

What backups enable

1. **Crash-safe resume.** If the executor dies mid-migration (process killed, network drop, OOM), re-running the same command with the same `--backup-dir` reads the header file, detects partial progress, and resumes from the last completed batch instead of re-quantizing the keys that already converted successfully. If the header is already completed, the executor only treats it as a no-op resume when the live index already matches the target schema. If the live index has been rolled back to the source schema, the completed backup is stale for the new run and the executor creates a fresh backup.
2. **Manual rollback.** The data file contains the original pre-quantization vector bytes. After a migration, you can use the rollback CLI (`rvl migrate rollback`) or the Python API to restore those bytes if you need to back out the change.

Retention

Backup files are **retained on disk** after a successful migration. Cleanup is now a deliberate operator action, performed only after the new vectors have been verified and rollback is no longer needed. Delete the backup directory manually when you are done.

Shared keys and overlapping indexes

Hash vector quantization rewrites the vector bytes stored in the Redis document key. It is supported only when the documents being quantized are not also indexed by another live RediSearch index that still expects the old vector datatype.

If the same Redis key is covered by multiple indexes, quantizing it for one index mutates the bytes seen by all other indexes. Those other indexes are not migrated at the same time, so the document can disappear from those indexes or fail to re-index because the stored vector bytes no longer match their schemas. The migrator does not support this topology for hash vector datatype changes.

Before applying a quantization migration, verify that the migrating index's keyspace is exclusive for the vector field being changed. If documents must be searchable through multiple indexes, use a coordinated application-level migration instead: create new physical keys or new vector fields, migrate every affected index schema together, and then switch traffic after validation.

For batch migrations, `batch-plan` performs a conservative prefix overlap check across every applicable index. Two indexes whose key prefixes overlap (one prefix is a literal string-prefix of the other, matching `FT.CREATE PREFIX` semantics) are refused because a batch quantization migration could re-read vectors that an earlier index in the batch has already quantized. The error names the conflicting indexes and the specific prefix pairs that overlap.

The batch overlap check is plan-time only — no data is mutated when a batch is refused. Resolve by splitting the indexes into prefix-disjoint groups and creating one batch plan per group. Indexes that are skipped for other reasons (e.g. `applicable: false` because a field is missing) do not participate in the check.

Why some changes are blocked

Vector dimension changes

Vector dimensions are determined by your embedding model. A 384 dimensional vector from one model is mathematically incompatible with a 768 dimensional index expecting vectors from a different model. There is no way to resize an embedding.

Resolution: Re-embed your documents using the new model and load them into a new index.

Storage type changes

Hash and JSON have different data layouts. Hash stores flat key value pairs. JSON stores nested structures. Converting between them requires understanding your schema and restructuring each document.

Resolution: Export your data, transform it to the new format, and reload into a new index.

Adding a vector field

Adding a vector field means all existing documents need vectors for that field. The migrator cannot generate these vectors because it does not know which embedding model to use or what content to embed.

Resolution: Add vectors to your documents using your application, then run the migration.

Downtime considerations

With `drop_recreate`, your index is unavailable between the drop and when re-indexing completes.

CRITICAL: Downtime requires both reads AND writes to be paused:

Requirement	Reason
Pause reads	Index is unavailable during migration
Pause writes	Redis updates indexes synchronously. Writes during migration may conflict with vector re-encoding or be missed

Plan for:

- Search unavailability during the migration window
- Partial results while indexing is in progress
- Resource usage from the re-indexing process
- Quantization time if changing vector datatypes

The duration depends on document count, field count, and vector dimensions. For large indexes, consider running migrations during low traffic periods.

Sync vs async execution

The migrator provides both synchronous and asynchronous execution modes.

What becomes async and what stays sync

The migration workflow has distinct phases. Here is what each mode affects:

Phase	Sync mode	Async mode	Notes
Plan generation	<code>MigrationPlanner.create_plan()</code>	<code>AsyncMigrationPlanner.create_plan()</code>	Reads index metadata from Redis
Schema snapshot	Sync Redis calls	Async Redis calls	Single <code>FT.INFO</code> command
Enumeration	<code>FT.AGGREGATE</code> (or <code>SCAN</code> fallback)	<code>FT.AGGREGATE</code> (or <code>SCAN</code> fallback)	Before drop, only if quantization needed
Drop index	<code>index.delete()</code>	<code>await index.delete()</code>	Single <code>FT.DROPINDEX</code> command
Quantization	Sequential <code>HGET</code> + <code>HSET</code>	Sequential <code>HGET</code> + batched <code>HSET</code>	Uses pre-enumerated keys
Create index	<code>index.create()</code>	<code>await index.create()</code>	Single <code>FT.CREATE</code> command
Readiness polling	<code>time.sleep()</code> loop	<code>asyncio.sleep()</code> loop	Polls <code>FT.INFO</code> until indexed
Validation	Sync Redis calls	Async Redis calls	Schema and doc count checks
CLI interaction	Always sync	Always sync	User prompts, file I/O
YAML read/write	Always sync	Always sync	Local filesystem only

When to use sync (default)

Sync execution is simpler and sufficient for most migrations:

- Small to medium indexes (under 100K documents)
- Index-only changes (algorithm, distance metric, field options)
- Interactive CLI usage where blocking is acceptable

For migrations without quantization, the Redis operations are fast single commands. Sync mode adds no meaningful overhead.

When to use async

Async execution (`--async` flag) provides benefits in specific scenarios:

Large quantization jobs (1M+ vectors)

Converting `float32` to `float16` requires reading every vector, converting it, and writing it back. The async executor:

- Enumerates documents using `FT.AGGREGATE WITHCOURSE` for index-specific enumeration (falls back to `SCAN` only if indexing failures exist)
- Pipelines `HSET` operations in batches (100-1000 operations per pipeline is optimal for Redis)

- Yields to the event loop between batches so other tasks can proceed

Large keyspaces (40M+ keys)

When your Redis instance has many keys and the index has indexing failures (requiring SCAN fallback), async mode yields between batches.

Async application integration

If your application uses asyncio, you can integrate migration directly:

```
import asyncio
from redisvl.migration import AsyncMigrationPlanner, AsyncMigrationExecutor

async def migrate():
    planner = AsyncMigrationPlanner()
    plan = await planner.create_plan("myindex", redis_url="redis://localhost:6379")

    executor = AsyncMigrationExecutor()
    report = await executor.apply(
        plan,
        redis_url="redis://localhost:6379",
        backup_dir="/tmp/migration_backups",
    )

asyncio.run(migrate())
```

Why async helps with quantization

The migrator uses an optimized enumeration strategy:

1. **Index-based enumeration:** Uses `FT.AGGREGATE WITHCURSOR` to enumerate only indexed documents (not the entire keyspace)
2. **Fallback for safety:** If the index has indexing failures (`hash_indexing_failures > 0`), falls back to `SCAN` to ensure completeness
3. **Enumerate before drop:** Captures the document list while the index still exists, then drops and quantizes

This optimization provides 10-1000x speedup for sparse indexes (where only a small fraction of prefix-matching keys are indexed).

Sync quantization:

```
enumerate keys (FT.AGGREGATE or SCAN) -> store list
for each batch of 500 keys:
    for each key:
        HGET field (blocks)
        convert array
        pipeline.HSET(field, new_bytes)
    pipeline.execute() (blocks)
```

Async quantization:

```
enumerate keys (FT.AGGREGATE or SCAN) -> store list
for each batch of 500 keys:
    for each key:
```

(continues on next page)

(continued from previous page)

```

    await HGET field (yields)
    convert array
    pipeline.HSET(field, new_bytes)
    await pipeline.execute() (yields)

```

Each `await` is a yield point where other coroutines can run. For millions of vectors, this prevents your application from freezing.

What async does NOT improve

Async execution does not reduce:

- **Total migration time:** Same work, different scheduling
- **Redis server load:** Same commands execute on the server
- **Downtime window:** Index remains unavailable during rebuild
- **Network round trips:** Same number of Redis calls

The benefit is application responsiveness, not faster migration.

Learn more

- *Migration guide*: Step by step instructions
- *Search and indexing*: How Redis Search indexes work

1.1.4 Field Attributes

Field attributes customize how Redis indexes and searches your data. Each field type has specific attributes that control indexing behavior, search capabilities, and storage options.

Common Attributes

These attributes are available on most non-vector field types (text, tag, numeric, geo).

sortable

Enables sorting results by this field. Without `sortable`, you cannot use the field in `ORDER BY` clauses.

Trade-off: Sortable fields consume additional memory to maintain a sorted index.

```

# YAML
- name: created_at
  type: numeric
  attrs:
    sortable: true

```

```

# Python
{"name": "created_at", "type": "numeric", "attrs": {"sortable": True}}

```

Use when: You need to sort search results by this field (e.g., “newest first”, “highest price”).

no_index

Stores the field without indexing it. The field value is available in search results but cannot be used in queries or filters.

Important: `no_index` only makes sense when combined with `sortable: true`. A field that is neither indexed nor sortable serves no purpose in the schema.

```
# YAML - Store for sorting but don't index for search
- name: internal_score
  type: numeric
  attrs:
    sortable: true
    no_index: true
```

Use when: You want to sort by a field but never filter on it, saving index space.

index_missing

Allows searching for documents that don't have this field. When enabled, you can use `ISMISSING` queries to find documents where the field is absent or null.

```
# YAML
- name: optional_category
  type: tag
  attrs:
    index_missing: true
```

```
# Python
{"name": "optional_category", "type": "tag", "attrs": {"index_missing": True}}
```

Use when: Your data has optional fields and you need to query for documents missing those fields.

Query example:

```
from redisvl.query.filter import Tag

# Find documents where category is missing
filter_expr = Tag("optional_category").ismissing()
```

Text Field Attributes

Text fields support full-text search with these additional attributes.

weight

Controls the importance of this field in relevance scoring. Higher weights make matches in this field rank higher.

```

- name: title
  type: text
  attrs:
    weight: 2.0 # Title matches count double

- name: description
  type: text
  attrs:
    weight: 1.0 # Default weight

```

Use when: Some text fields are more important than others for search relevance.

no_stem

Disables stemming for this field. By default, Redis applies stemming so “running” matches “run”. Disable when exact word forms matter.

```

- name: product_code
  type: text
  attrs:
    no_stem: true

```

Use when: Field contains codes, identifiers, or technical terms where stemming would cause incorrect matches.

withsuffixtrie

Maintains a suffix trie for optimized suffix and contains queries. Enables efficient `*suffix` and `*contains*` searches.

```

- name: email
  type: text
  attrs:
    withsuffixtrie: true

```

Use when: You need to search for patterns like `*@gmail.com` or `*smith*`.

Trade-off: Increases memory usage and index build time.

phonetic_matcher

Enables phonetic matching using the specified algorithm. Matches words that sound similar.

```

- name: name
  type: text
  attrs:
    phonetic_matcher: "dm:en" # Double Metaphone, English

```

Supported values: `dm:en` (Double Metaphone English), `dm:fr` (French), `dm:pt` (Portuguese), `dm:es` (Spanish)

Use when: Searching names or words where spelling variations should match (e.g., “Smith” matches “Smyth”).

index_empty

Allows indexing and searching for empty strings. By default, empty strings are not indexed.

```
- name: middle_name
  type: text
  attrs:
    index_empty: true
```

Use when: Empty string is a meaningful value you need to query for.

unf (Un-Normalized Form)

Preserves the original value for sortable fields without normalization. By default, sortable text fields are lowercased for consistent sorting.

Requires: sortable: true

```
- name: title
  type: text
  attrs:
    sortable: true
    unf: true # Keep original case for sorting
```

Use when: You need case-sensitive sorting or must preserve exact original values.

Tag Field Attributes

Tag fields are for exact-match filtering on categorical data.

separator

Specifies the character that separates multiple tags in a single field value. Default is comma (,).

```
- name: categories
  type: tag
  attrs:
    separator: "|" # Use pipe instead of comma
```

Use when: Your tag values contain commas, or you're using a different delimiter in your data.

case_sensitive

Makes tag matching case-sensitive. By default, tags are lowercased for matching.

```
- name: product_sku
  type: tag
  attrs:
    case_sensitive: true
```

Use when: Tag values are case-sensitive identifiers (SKUs, codes, etc.).

withsuffixtrie

Same as text fields—enables efficient suffix and contains queries on tags.

```
- name: email_domain
  type: tag
  attrs:
    withsuffixtrie: true
```

index_empty

Allows indexing empty tag values.

```
- name: optional_tags
  type: tag
  attrs:
    index_empty: true
```

Numeric Field Attributes

Numeric fields support range queries and sorting.

unf (Un-Normalized Form)

For sortable numeric fields, preserves the exact numeric representation without normalization.

Requires: sortable: true

```
- name: price
  type: numeric
  attrs:
    sortable: true
    unf: true
```

Note: Numeric fields do not support `index_empty` (empty numeric values are not meaningful).

Geo Field Attributes

Geo fields store geographic coordinates for location-based queries.

Geo fields support the common attributes (`sortable`, `no_index`, `index_missing`) but have no geo-specific attributes. The field value should be a string in "longitude,latitude" format.

```
- name: location
  type: geo
  attrs:
    sortable: true
```

Note: Geo fields do not support `index_empty` (empty coordinates are not meaningful).

Vector Field Attributes

Vector fields have a different attribute structure. See [Schema](#) for complete vector field documentation.

Key vector attributes:

- `dims`: Vector dimensionality (required)
- `algorithm`: flat, hnsw, or svsvamana
- `distance_metric`: COSINE, L2, or IP
- `datatype`: Vector precision (see table below)
- `index_missing`: Allow searching for documents without vectors

```
- name: embedding
  type: vector
  attrs:
    algorithm: hnsw
    dims: 768
    distance_metric: cosine
    datatype: float32
    index_missing: true # Handle documents without embeddings
```

Vector Datatypes

The datatype attribute controls how vector components are stored. Smaller datatypes reduce memory usage but may affect precision.

Datatype	Bits	Memory (768 dims)	Use Case
float32	32	3 KB	Default. Best precision for most applications.
float16	16	1.5 KB	Good balance of memory and precision. Recommended for large-scale deployments.
bfloat16	16	1.5 KB	Better dynamic range than float16. Useful when embeddings have large value ranges.
float64	64	6 KB	Maximum precision. Rarely needed.
int8	8	768 B	Integer quantization. Significant memory savings with some precision loss.
uint8	8	768 B	Unsigned integer quantization. For embeddings with non-negative values.

Algorithm Compatibility:

Datatype	FLAT	HNSW	SVS-VAMANA
float32	Yes	Yes	Yes
float16	Yes	Yes	Yes
bfloat16	Yes	Yes	No
float64	Yes	Yes	No
int8	Yes	Yes	No
uint8	Yes	Yes	No

Choosing a Datatype:

- **Start with float32** unless you have memory constraints
- **Use float16** for production systems with millions of vectors (50% memory savings, minimal precision loss)
- **Use int8/uint8** only after benchmarking recall on your specific dataset
- **SVS-VAMANA users:** Must use float16 or float32

Quantization with the Migrator:

You can change vector datatypes on existing indexes using the migration wizard:

```
rvl migrate wizard --index my_index --url redis://localhost:6379
# Select "Update field" > choose vector field > change datatype
```

The migrator automatically re-encodes stored vectors to the new precision. See [Migrate an Index](#) for details. When you apply the resulting migration plan, pass `--backup-dir`; the backup directory is required before any migration starts and stores original vector bytes for resume and rollback.

Redis-Specific Subtleties

Modifier Ordering

Redis Search has specific requirements for the order of field modifiers. RedisVL handles this automatically, but it's useful to understand:

Canonical order: INDEXEMPTY → INDEXMISSING → SORTABLE → UNF → NOINDEX

If you're debugging raw Redis commands, ensure modifiers appear in this order.

Field Type Limitations

Not all attributes work with all field types:

Attribute	Text	Tag	Numeric	Geo	Vector
sortable	✓	✓	✓	✓	
no_index	✓	✓	✓	✓	
index_missing	✓	✓	✓	✓	✓
index_empty	✓	✓			
unf	✓		✓		
withsuffixtrie	✓	✓			

Migration Support

The migration wizard (`rvl migrate wizard`) supports updating field attributes on existing indexes. The table below shows which attributes can be updated via the wizard vs requiring manual schema patch editing.

Wizard Prompts:

Attribute	Text	Tag	Numeric	Geo	Vector
sortable	Wizard	Wizard	Wizard	Wizard	N/A
index_missing	Wizard	Wizard	Wizard	Wizard	N/A
index_empty	Wizard	Wizard	N/A	N/A	N/A
no_index	Wizard	Wizard	Wizard	Wizard	N/A
unf	Wizard*	N/A	Wizard*	N/A	N/A
separator	N/A	Wizard	N/A	N/A	N/A
case_sensitive	N/A	Wizard	N/A	N/A	N/A
no_stem	Wizard	N/A	N/A	N/A	N/A
weight	Wizard	N/A	N/A	N/A	N/A
algorithm	N/A	N/A	N/A	N/A	Wizard
datatype	N/A	N/A	N/A	N/A	Wizard
distance_metric	N/A	N/A	N/A	N/A	Wizard
m_ef_construction	N/A	N/A	N/A	N/A	Wizard

* *unf* is only prompted when *sortable* is enabled.

Manual Schema Patch Required:

Attribute	Notes
withsuffixtrie	Suffix/contains search optimization

Note: *phonetic_matcher* is supported by the wizard for text fields.

Example manual patch for adding `index_missing` to a field:

```
# schema_patch.yaml
version: 1
changes:
  update_fields:
    - name: category
      attrs:
        index_missing: true
```

```
rvl migrate plan --index my_index --schema-patch schema_patch.yaml
```

JSON Path for Nested Fields

When using JSON storage, use the `path` attribute to index nested fields:

```
- name: author_name
  type: text
  path: $.metadata.author.name
  attrs:
    sortable: true
```

The name becomes the field's alias in queries, while `path` specifies where to find the data.

Complete Example

```
version: "0.1.0"
index:
  name: products
  prefix: product
  storage_type: json

fields:
  # Full-text searchable with high relevance
  - name: title
    type: text
    path: $.title
    attrs:
      weight: 2.0
      sortable: true

  # Exact-match categories
  - name: category
    type: tag
    path: $.category
    attrs:
      separator: "|"
      index_missing: true

  # Sortable price with range queries
  - name: price
    type: numeric
    path: $.price
    attrs:
      sortable: true

  # Store-only field for sorting
  - name: internal_rank
    type: numeric
    path: $.internal_rank
    attrs:
      sortable: true
      no_index: true

  # Vector embeddings
  - name: embedding
    type: vector
    path: $.embedding
    attrs:
      algorithm: hnsw
      dims: 768
      distance_metric: cosine

  # Location search
  - name: store_location
    type: geo
    path: $.location
```

Learn more: [Schema](#) provides the complete API reference for all field types and attributes.

1.1.5 Query Types

RedisVL provides several query types, each optimized for different search patterns. Understanding when to use each helps you build efficient search applications.

Vector Queries

Vector queries find documents by semantic similarity. You provide a vector (typically an embedding of text or images), and Redis returns documents whose vectors are closest to yours.

VectorQuery

The most common query type. Returns the top K most similar documents using KNN (K-Nearest Neighbors) search.

```
from redisvl.query import VectorQuery

query = VectorQuery(
    vector=embedding,           # Your query embedding
    vector_field_name="embedding",
    num_results=10
)
results = index.query(query)
```

Use when you want to find the N most similar items regardless of how similar they actually are. Good for “find me things like this” searches.

VectorRangeQuery

Returns all documents within a specified distance threshold. Unlike VectorQuery, this doesn’t limit results to a fixed K—it returns everything within the radius.

```
from redisvl.query import VectorRangeQuery

query = VectorRangeQuery(
    vector=embedding,
    vector_field_name="embedding",
    distance_threshold=0.3      # Return all within this distance
)
results = index.query(query)
```

Use when similarity threshold matters more than result count. Good for “find everything similar enough” searches, like deduplication or clustering.

Filter Queries

Filter queries find documents by exact field matching without vector similarity.

FilterQuery

Searches using filter expressions on indexed fields. Supports tag matching, numeric ranges, text search, and geographic filters.

```
from redisvl.query import FilterQuery
from redisvl.query.filter import Tag, Num

query = FilterQuery(
    filter_expression=(Tag("category") == "electronics") & (Num("price") < 100),
    return_fields=["title", "price"],
    num_results=20
)
results = index.query(query)
```

Use when you need precise filtering without semantic similarity—finding all products in a category, all users in a region, or all records within a date range.

CountQuery

Returns only the count of matching documents, not the documents themselves. More efficient than FilterQuery when you only need the count.

```
from redisvl.query import CountQuery
from redisvl.query.filter import Tag

query = CountQuery(filter_expression=Tag("status") == "active")
count = index.query(query)
```

Use for analytics, pagination totals, or checking if matches exist before running a full query.

Text Queries

Text queries perform full-text search with relevance scoring.

TextQuery

Searches text fields using Redis's full-text search capabilities. Supports multiple scoring algorithms (BM25, TF-IDF), stopword handling, and field weighting.

```
from redisvl.query import TextQuery

query = TextQuery(
    text="machine learning",
    text_field_name="content",
    text_scorer="BM25STD",
```

(continues on next page)

(continued from previous page)

```
        num_results=10
    )
    results = index.query(query)
```

Use when you need keyword-based search with relevance ranking—traditional search engine behavior where exact word matches matter.

Hybrid Queries

Hybrid queries combine multiple search strategies for better results than either alone.

HybridQuery

Combines text search and vector search in a single query using Redis's native hybrid search. Supports multiple fusion methods:

- **RRF (Reciprocal Rank Fusion)**: Combines rankings from both searches. Good when you trust both signals equally.
- **Linear**: Weighted combination of scores. Good when you want to tune the balance between text and semantic relevance.

```
from redisvl.query import HybridQuery

query = HybridQuery(
    text="machine learning frameworks",
    text_field_name="content",
    vector=embedding,
    vector_field_name="embedding",
    combination_method="RRF",
    num_results=10
)
results = index.query(query)
```

Use when neither pure keyword search nor pure semantic search gives good enough results. Common in RAG applications where you want both exact matches and semantic understanding.

Note

HybridQuery requires Redis $\geq$ 8.4.0 and redis-py $\geq$ 7.1.0.

AggregateHybridQuery

Similar to HybridQuery but uses Redis aggregation pipelines. Provides more control over score combination and result processing.

Use when you need custom score normalization or complex result transformations that HybridQuery doesn't support.

Multi-Vector Queries

MultiVectorQuery

Searches across multiple vector fields simultaneously with configurable weights per field.

```
from redisvl.query import MultiVectorQuery, Vector

query = MultiVectorQuery(
    vectors=[
        Vector(vector=text_embedding, field_name="text_vector", weight=0.7),
        Vector(vector=image_embedding, field_name="image_vector", weight=0.3),
    ],
    num_results=10
)
results = index.query(query)
```

Use for multimodal search—finding documents that match across text embeddings, image embeddings, and other vector representations. Each vector field can have different importance weights.

SQL Queries

SQLQuery

Translates SQL SELECT statements into Redis queries. Provides a familiar interface for developers coming from relational databases.

```
from redisvl.query import SQLQuery

query = SQLQuery("""
    SELECT title, price, category
    FROM products
    WHERE category = 'electronics' AND price < 100
""")
results = index.query(query)
```

SQLQuery also accepts `sql_redis_options`, which are forwarded to the underlying `sql-redis` executor. This is mainly useful for tuning schema caching behavior.

```
query = SQLQuery(
    """
    SELECT title, price, category
    FROM products
    WHERE category = 'electronics' AND price < 100
    """,
```

(continues on next page)

(continued from previous page)

```
sql_redis_options={"schema_cache_strategy": "lazy"},
)
```

- "lazy" (default) loads schemas only when a query touches an index, which keeps startup and one-off queries cheaper.
- "load_all" preloads all schemas up front, which can help repeated query workloads that span many indexes.

For TEXT fields with `sql-redis` $\geq$ 0.4.0:

- `=` performs exact phrase or exact-term matching
- `LIKE` performs prefix/suffix/contains matching using SQL % wildcards
- `fuzzy(field, 'term')` performs typo-tolerant matching
- `fulltext(field, 'query')` performs tokenized search

```
query = SQLQuery("SELECT * FROM products WHERE title = 'gaming laptop'")
query = SQLQuery("SELECT * FROM products WHERE title LIKE 'lap%'")
query = SQLQuery("SELECT * FROM products WHERE fuzzy(title, 'laptap'")
query = SQLQuery("SELECT * FROM products WHERE fulltext(title, 'laptop OR tablet'")
```

Use `=` when you want an exact phrase, `LIKE` for prefix/suffix/contains patterns, `fuzzy()` for typo-tolerant lookup, and `fulltext()` for tokenized search operators such as `OR`, optional terms, or proximity.

Aggregations and grouping:

```
query = SQLQuery("""
    SELECT category, COUNT(*) as count, AVG(price) as avg_price
    FROM products
    GROUP BY category
    ORDER BY count DESC
""")
```

Geographic queries with `geo_distance()`:

```
# Find stores within 50km of San Francisco
query = SQLQuery("""
    SELECT name, category
    FROM stores
    WHERE geo_distance(location, POINT(-122.4194, 37.7749), 'km') < 50
""")

# Calculate distances
query = SQLQuery("""
    SELECT name, geo_distance(location, POINT(-122.4194, 37.7749)) AS distance
    FROM stores
""")
```

Date queries with ISO date literals and date functions:

```
# Filter by date range
query = SQLQuery("""
    SELECT name FROM events
    WHERE created_at BETWEEN '2024-01-01' AND '2024-03-31'
""")
```

(continues on next page)

(continued from previous page)

```

"""
# Extract date parts
query = SQLQuery("""
    SELECT YEAR(created_at) AS year, COUNT(*) AS count
    FROM events
    GROUP BY year
""")

```

Vector similarity search with parameters:

```

query = SQLQuery("""
    SELECT title, vector_distance(embedding, :vec) AS score
    FROM products
    LIMIT 5
""", params={"vec": embedding_bytes})

```

Hybrid search (FT.HYBRID) fuses a text query and a vector query server-side with `hybrid_vector_search()`, composing `cosine_distance()` (vector leg) and `fulltext()` (text leg) with `rrf()` or `linear()` fusion. This is the SQL front-end to the native HybridQuery (above):

```

query = SQLQuery("""
    SELECT title,
        hybrid_vector_search(
            cosine_distance(embedding, :vec),
            fulltext(description, 'gaming laptop'),
            rrf()
        ) AS hybrid_score
    FROM products
    ORDER BY hybrid_score DESC
    LIMIT 5
""", params={"vec": embedding_bytes})

```

Requires Redis 8.4+ and `redis-py >= 7.1.0`.

Use when your team is more comfortable with SQL syntax, or when integrating with tools that generate SQL.

Note

SQLQuery requires the optional `sql-redis` package. Install with: `pip install redisvl[sql-redis]`

For comprehensive examples including geographic filtering, date functions, and vector search, see the [SQL to Redis Queries guide](#).

Choosing the Right Query

Use Case	Query Type
Semantic similarity search	VectorQuery
Find all items within similarity threshold	VectorRangeQuery
Exact field matching	FilterQuery
Count matching records	CountQuery
Keyword search with relevance	TextQuery
Combined keyword + semantic	HybridQuery
Multimodal search	MultiVectorQuery
SQL-familiar interface	SQLQuery

Common Patterns

Vector Search with Filters

All vector queries support filter expressions. Combine semantic search with metadata filtering:

```
from redisvl.query import VectorQuery
from redisvl.query.filter import Tag, Num

query = VectorQuery(
    vector=embedding,
    vector_field_name="embedding",
    filter_expression=(Tag("category") == "electronics") & (Num("price") < 100),
    num_results=10
)
```

Hybrid Search for RAG

For retrieval-augmented generation, hybrid search often outperforms pure vector search:

```
from redisvl.query import HybridQuery

query = HybridQuery(
    text="machine learning frameworks",
    text_field_name="content",
    vector=embedding,
    vector_field_name="embedding",
    combination_method="RRF",
    num_results=5
)
```

Results and expiring documents

RedisVL returns query results as a list of dictionaries, one per matched document. On Redis 8.8+ a document can occasionally come back with its field payload missing: the RediSearch worker pool now defaults to a nonzero number of background worker threads, and if a document — or one of its indexed fields — expires (via TTL or HPEXPIRE) at the exact moment a background search is reading it, the server returns the matched id with an empty field payload instead of dropping it from the result set. This is most likely in workloads that combine search with short TTLs, such as a semantic cache or chat history.

RedisVL detects and **silently skips** such a document (logging at WARNING level) whenever the query type guarantees a field would be present on a healthy match:

- **VectorQuery / VectorRangeQuery** with `return_score=True` (the default) — a healthy match always carries a `vector_distance`.
- **FilterQuery on JSON storage** that returns the whole object — a healthy match always carries its JSON payload.

For a plain **FilterQuery / TextQuery on hash storage**, a race victim is indistinguishable from a legitimately sparse result (for example, a document matched via an `INDEXMISSING` field, or a query that requested only the `id`). RedisVL cannot safely tell those apart, so it returns the document as an id-only `{"id": ...}` dict rather than risk dropping a valid result. If you query hash storage directly this way, use `.get()` for optional fields rather than assuming every field is present.

Detecting incomplete results

`SearchIndex.query()` returns a `SearchResults` object, which is a normal list of result dictionaries with two extra attributes so you can tell a race-shortened result set apart from one that genuinely had fewer matches:

- `results.dropped_count` — how many matched documents were skipped because their field payload was missing (0 in the normal case).
- `results.complete` — False when any document was dropped.

```
results = index.query(query)

if not results.complete:
    # e.g. retry, reconcile against a CountQuery, or annotate the answer
    logger.warning("Result set is incomplete: %d dropped", results.dropped_count)
```

`SearchResults` behaves exactly like a list everywhere else, so existing code needs no changes. (Operations that build a new list — slicing, `sorted()`, concatenation — return a plain list without these attributes.)

Practical consequences:

- A query may return **fewer results than `num_results`** (or fewer than `page_size` when paginating) when some matched documents were expiring. Treat those limits as upper bounds, not guarantees; use `results.complete` to detect when it happened.
- A `CountQuery` reports the server's match count, which still includes the expiring document, so a count can legitimately exceed the number of documents a materializing query returns at the same instant.
- **`paginate()` keeps going past a page it could not materialize.** If every match on one page was expiring, that page is skipped and iteration continues to the end of the offset range rather than stopping early. An empty batch is never yielded, and a skipped page's `dropped_count` is added to the next batch you receive, so `results.complete` stays meaningful across the whole stream. Two caveats: a result set whose *trailing* pages were entirely dropped has no later batch to report on, so those drops appear only in the logged warning; and paging is only

stable if the query sorts on a unique field — Redis documents `LIMIT` without sorting as non-deterministic, which can duplicate or miss documents independently of expiry.

- The higher-level extensions build on this: the **semantic cache** drops an expiring hit (a benign cache miss), **message history** drops an expiring message from its formatted output, and the **semantic router** drops an expiring route candidate (which, in the rare case the best match is the one expiring, can shift the selected route). Message history's `raw=True` mode returns the unprocessed hash entries and may therefore still surface an id-only record.

Learn more: *Use Advanced Query Types* demonstrates these query types in detail.

1.1.6 Utilities

Utilities are optional components that enhance search workflows. They're not required—you can bring your own embeddings and skip reranking—but they simplify common tasks.

Vectorizers

A vectorizer converts text into an embedding vector. Embeddings are dense numerical representations that capture semantic meaning: similar texts produce similar vectors, enabling similarity search.

Why Vectorizers Matter

Creating embeddings requires calling an embedding model—either a cloud API (OpenAI, Cohere, etc.) or a local model (HuggingFace sentence-transformers). Each provider has different APIs, authentication methods, and response formats.

RedisVL vectorizers provide a unified interface across providers. You choose a provider, configure it once, and use the same methods regardless of which service is behind it. This makes it easy to switch providers, compare models, or use different providers in different environments.

The Dimensionality Contract

Every embedding model produces vectors of a specific size (dimensionality). OpenAI's text-embedding-3-small produces 1536-dimensional vectors. Other models produce 384, 768, 1024, or other sizes.

Your schema's vector field must specify the same dimensionality as your embedding model. If there's a mismatch—your model produces 1536-dimensional vectors but your schema expects 768—you'll get errors when loading data or running queries.

This constraint means you should choose your embedding model before designing your schema, and changing models requires rebuilding your index.

Batching and Performance

Embedding APIs have rate limits and per-request overhead. Embedding one text at a time is inefficient. Vectorizers support batch embedding, sending multiple texts in a single request. This dramatically improves throughput for indexing large datasets.

Vectorizers handle batching internally, breaking large batches into provider-appropriate chunks and respecting rate limits. You provide a list of texts; the vectorizer manages the logistics.

Supported Providers

RedisVL includes vectorizers for OpenAI, Azure OpenAI, Cohere, HuggingFace (local), Mistral, Google (via `google-genai`, covering both Vertex AI and the Gemini Developer API), AWS Bedrock, VoyageAI, and others. See the [Vectorizers](#) for the complete list. You can also create custom vectorizers that wrap any embedding function.

Rerankers

A reranker takes initial search results and reorders them by relevance to the query. It's a second-stage filter that improves precision after the first-stage retrieval.

Why Reranking Works

Vector search uses bi-encoder models: the query and documents are embedded independently, then compared by vector distance. This is fast but approximate—the embedding captures general meaning, not the specific relationship between query and document.

Rerankers use cross-encoder models that score query-document pairs directly. The model sees both the query and document together and predicts a relevance score. This is more accurate but slower, because each candidate requires a separate model inference.

The combination is powerful: use fast vector search to retrieve a broad set of candidates (high recall), then use the slower but more accurate reranker to select the best results (high precision).

The Recall-Precision Trade-off

With only vector search, you might retrieve 10 results and hope the best one is in there. With reranking, you can retrieve 50 candidates—casting a wider net—then rerank to find the 5 best. The initial retrieval prioritizes recall (not missing relevant documents); reranking prioritizes precision (surfacing the most relevant ones).

Cost and Latency

Reranking adds latency (typically 50-200ms depending on the provider and number of candidates) and cost (API-based rerankers charge per request). These trade-offs are usually worthwhile when result quality matters, but you should measure the impact for your use case.

Supported Providers

RedisVL includes rerankers for HuggingFace cross-encoders (local), Cohere Rerank API, and VoyageAI Rerank API.

Two-Stage Retrieval

The most effective retrieval pipelines combine both utilities: vectorize the query, retrieve a candidate set with vector search, then rerank to select the final results.

This pattern separates recall (finding everything potentially relevant) from precision (selecting the best matches). Vector search handles recall efficiently; reranking handles precision accurately. Together, they deliver better results than either approach alone.

Related concepts: *Query Types* explains how to use embeddings in vector search queries. *Search & Indexing* covers schema configuration for vector fields.

Learn more: *Create Embeddings with Vectorizers* covers embedding providers. *Rerank Search Results* explains reranking in practice.

1.1.7 RedisVL MCP

RedisVL includes an MCP server that exposes a Redis-backed retrieval surface through a small, deterministic tool contract. It is designed for AI applications that want to search or maintain data in one or more existing Redis indexes without each client reimplementing Redis query logic.

What RedisVL MCP Does

The RedisVL MCP server sits between an MCP client and Redis:

1. It connects to one or more existing Redis Search indexes.
2. It inspects each index at startup and reconstructs its schema.
3. It initializes vector capabilities only when the configured search or upsert behavior needs them.
4. It exposes stable MCP tools for discovery, search, and optionally upsert.

This keeps each Redis index as the source of truth for its search behavior while giving MCP clients a predictable interface.

How RedisVL MCP Runs

RedisVL MCP works with a focused model:

- One server process binds to one *or several* existing Redis indexes, each addressed by a logical id.
- The server supports stdio (default), Streamable HTTP, and SSE transports.
- Search behavior is owned by per-index configuration, not by MCP callers.
- Vector search and server-side embedding are optional capabilities configured explicitly per index.
- Upsert is optional and can be disabled globally with read-only mode or per index with a `read_only` flag.

A single-index server remains the simplest deployment: when exactly one index is configured, callers can omit the index selector entirely and every tool call targets that index. Multi-index support is fully formal — it adds discovery and explicit routing without changing the single-index contract.

Config-Owned Search Behavior

MCP callers can control:

- `query`
- `limit`
- `offset`
- `filter`
- `return_fields`

These request-time controls are still bounded by runtime config. In particular, deep paging is limited by a configured maximum result window, enforced as `offset + limit`.

On a multi-index server, callers also choose **which index to target** through an optional `index` argument (see [Index Selection](#)). Callers do not choose:

- whether retrieval is `vector`, `fulltext`, or `hybrid`
- query tuning parameters such as hybrid fusion or vector runtime settings

That behavior lives in the per-index server config under `indexes.<id>.search`. The response includes `search_type` as informational metadata, but it is not a request parameter.

Single and Multiple Index Bindings

The YAML config uses an `indexes` mapping. Each entry is a logical binding keyed by an id (for example `knowledge` or `tickets`) that points to an existing Redis index through `redis_name`. The mapping may contain one entry or several; each binding is inspected, validated, and given its own search config, runtime limits, and optional vectorizer independently at startup. Startup is all-or-nothing — if any binding fails to initialize, the server does not start.

A single-binding config is the simplest case and behaves exactly as before: the lone binding is the implicit target of every call. With multiple bindings the server stays a single process and endpoint, but callers select a binding per call.

Index Selection and Discovery

On a multi-index server, every tool call must say which logical index it targets:

- `search-records` and `upsert-records` accept an optional `index` argument naming the logical id.
- When exactly one index is configured, `index` may be omitted and resolves to that sole binding (backward compatible).
- When multiple indexes are configured, omitting `index` is an `invalid_request`; the caller must name one.
- An unknown logical id is an `invalid_request`.
- Both tools echo the resolved `index` in their response so clients can confirm routing.

Because a client cannot guess the configured logical ids, multi-index servers expose a `list-indexes` discovery tool. **Clients should call `list-indexes` first** to enumerate the available indexes and their filterable fields, then pass the chosen id as `index` on subsequent calls.

Schema Inspection and Overrides

RedisVL MCP is inspection-first:

- the Redis index must already exist
- the server reconstructs the schema from Redis metadata at startup
- runtime field mappings remain explicit in config

In some environments, Redis metadata can be incomplete for vector field attributes. When that happens, `schema_overrides` can patch missing attrs for fields that were already discovered. It does not create new fields or change discovered field identity.

Startup also validates that the inspected schema does not collide with MCP-reserved score metadata field names for the configured search mode.

Read-Only and Read-Write Modes

RedisVL MCP registers `search-records` and `list-indexes` by default (see *Custom Tool Profiles* for turning a built-in off deliberately).

Write availability is enforced at two levels:

- **Global read-only mode** disables writes across every binding. It is controlled by the CLI flag `--read-only` or the environment variable `REDISVL_MCP_READ_ONLY=true`.
- **Per-index read-only** disables writes for a single binding via `indexes.<id>.read_only: true`, while other bindings stay writable.

These combine into each binding's *effective* write availability: a binding is read-only if global read-only is on **or** that binding sets `read_only: true`. The `upsert-records` tool is registered only when at least one binding is writable, so a fully read-only server does not advertise it at all. When the tool is registered, a write to a read-only binding is rejected with `forbidden` before any data is changed. `list-indexes` reports each binding's effective write availability as `upsert_available`.

Use read-only mode when Redis is serving approved content to assistants and another system owns ingestion — globally when no binding should accept writes, or per index when only some indexes are writable.

Authentication and Authorization

The HTTP transports can require a JWT bearer token issued by an existing identity provider. The server validates the token signature, issuer, and audience, and can gate read vs write by scope or role claim. This is coarse, per-tool authorization; it does not map token claims to Redis ACL users or per-tenant filters, which remain a gateway concern. The `stdio` transport is local and is never authenticated.

For configuration and the gateway boundary, see *Authenticate RedisVL MCP*.

Tool Surface

RedisVL MCP exposes up to three built-in tools, plus any configured *custom tool profiles*:

- `list-indexes` enumerates the configured logical indexes for discovery
- `search-records` searches a selected index using that index's server-owned search mode
- `upsert-records` validates and upserts records into a selected writable index, embedding them only when that capability is configured

Any of the three can be turned off with `server.builtin_tools`, independently of whether custom tools are configured — useful for a server that should only ever read, or one that serves nothing but curated profiles:

```
server:
  builtin_tools:
    upsert-records: disabled
```

Only the three names above are accepted; anything else fails at startup rather than being silently ignored.

Disabling a built-in adjusts what the rest of the surface advertises, so the published contract never points at something the server withholds:

- `list-indexes` reports `upsert_available: false` for every binding when `upsert-records` is disabled, since a writable binding still cannot be written to through a tool that is not published.
- On a multi-index server with `list-indexes` disabled, every tool that requires an `index` — `search-records` and `upsert-records` alike — names the available index ids in its own description instead of deferring to a discovery tool that does not exist. That server still logs a startup warning naming the affected tools, because inlining the ids is a fallback rather than an endorsement of the shape.

A server whose tool set ends up unusable — no tools at all, or discovery disabled on a multi-index server — logs a warning at startup.

Tools register once per process. `builtin_tools` is re-read on restart, but the registered tool set is not rebuilt, so a stop/start against an edited config keeps the previous tools and logs a warning saying so. Start a new process to change the tool surface.

These built-in tools follow a stable contract (profiles differ where noted in *Custom Tool Profiles* — notably they accept the object filter form only):

- request validation happens before query or write execution
- the resolved logical `index` is echoed in every `search-records` and `upsert-records` response
- filters support either raw strings or a RedisVL-backed JSON DSL
- on a single-index server, `search-records` describes the inspected schema by advertising typed JSON DSL filter fields, object-filter `exists` support, and valid `return_fields`; on a multi-index server those hints are ambiguous, so the description instead directs clients to call `list-indexes` and pass `index`
- error codes are mapped into a stable set of MCP-facing categories

list-indexes

list-indexes returns one entry per configured binding so clients can route subsequent calls. Each entry reports:

- the logical id
- an optional description (only when configured)
- upsert_available, reflecting the binding's effective write availability
- fields, the filterable fields discovered from the index
- limits, only the runtime limits that were explicitly configured

The discovery payload is deliberately minimal:

- the underlying Redis index name (redis_name) is **never** exposed
- the vector field and the configured embed-source text field are **omitted** from fields, since they are implementation inputs rather than fields a client filters on
- limits shows only explicitly set values (such as max_limit or max_upsert_records); defaults are not echoed

Custom Tool Profiles

The built-in tools expose the index generically, which leaves the model doing query engineering on every call: pick the index, understand the schema, build a filter, choose return fields. A **profile** moves those decisions into config. It is search-records with some arguments pre-filled and frozen and the rest still exposed, published under a name and description of your choosing.

Profiles are pure configuration. You add a custom_tools entry to the same YAML the server already loads and restart it; there is no Python to write.

```
custom_tools:
- name: search-support-tickets
  based_on: search-records
  index: support_tickets
  description: >
    Search historical customer support tickets by semantic similarity.
    Use this to find prior resolutions for a customer problem.
  lock:
    return_fields: [subject, resolution, created_at]
    filter: { field: status, op: eq, value: resolved }
  params:
    limit: { expose: true, max: 20 }
    filter: { expose: true }
```

lock holds what the author decides; params holds what the model may still pass. Anything not listed in params stays exposed, so a profile that only locks a filter keeps the rest of the built-in's contract. index is pinned by the top-level index: key rather than exposed as a param, and may be omitted only when exactly one index is configured.

params.limit.max bounds the result count. It applies whether the model names a limit or leaves it out — an omitted limit is capped rather than falling through to the binding's default. An explicit request above the cap is rejected. When limit is hidden (expose: false), the cap becomes the fixed result count instead. The cap must not exceed the binding's own runtime.max_limit, which is checked at startup. Note that it bounds page size, not total reachable data: offset is a separate argument, so paging is still possible up to the binding's max_result_window.

`suppress_schema_hints: true` drops the auto-generated field hints from the tool description. Those hints enumerate filterable and returnable fields, which is noise once those arguments are locked — and by default they are appended only for arguments the model can still use.

Two things about filters are easy to conflate:

- `lock.filter` is an ordinary expression in the JSON filter DSL. Its `and/or/not` operators describe the locked filter's own content.
- How the locked filter combines with a model-supplied one is fixed and not configurable: the executed query is always `locked AND caller`. A compound model-supplied expression renders parenthesized, so its `or/not` nests *inside* the locked `AND` and cannot reach the top level — the model can only narrow within the locked scope and never widen past it.

For that reason a profile accepts only the **object** form of a filter from the model. A raw filter string is rejected both by the advertised schema and by the tool itself, because strings bypass the DSL's field validation and have no safe composition with a locked expression.

Structure is only half of it: the nesting guarantee holds only while every filter *value* stays inside its own clause. Text `eq/ne` values are handled by the `Text` filter itself, which renders them as a quoted phrase with any `"` or `\` replaced by a space, so a parenthesis or a `|` the value carries is literal text rather than syntax. Text `like` values are patterns, so the library leaves them raw and this boundary escapes them instead — the delimiters that would close the clause are escaped, the pattern metacharacters are not. Tag values have their delimiters escaped and numeric values are type-checked. A caller filter that still renders as something able to break out is refused rather than combined.

Profiles resolve to a built-in call and nothing more, so they inherit the concurrency cap, request timeout, read-only policy, auth scoping, and error mapping already applied to `search-records`.

Because adding near-duplicate tools makes tool selection harder rather than easier, built-ins that curated profiles supersede can be turned off with `server.builtin_tools` (see [Tool Surface](#)).

Misconfiguration fails at startup rather than at the first call. Among the checks: a name colliding with a built-in or using a reserved `redisvl-/redisvl_` prefix; a duplicate tool name; a missing or unknown `index`; a `params` key that is not a real argument; `max` on anything but `limit`, or a cap above the binding's `max_limit`; hiding `query`; locking `return_fields` while also exposing them; and a locked filter or projection naming a field the bound index does not have. Unrecognized keys are rejected too, so a typo in `lock` fails loudly instead of silently producing a tool that reads as locked but enforces nothing.

Why Use MCP Instead of Direct RedisVL Calls

Use RedisVL MCP when you want a standard tool boundary for agent frameworks or assistants that already speak MCP.

Use direct RedisVL client code when your application should own index lifecycle, search construction, data loading, or richer RedisVL features directly in Python.

RedisVL MCP is a good fit when:

- multiple assistants should share one approved retrieval surface
- you want search behavior fixed by deployment config
- you need a read-only or tightly controlled write boundary
- you want to reuse an existing Redis index without rebuilding retrieval logic in every client

For setup steps, config, commands, and examples, see [Run RedisVL MCP](#).

1.1.8 Extensions

Extensions are opinionated, higher-level abstractions built on RedisVL's core primitives. Each extension encapsulates a common AI application pattern, managing its own Redis index internally and exposing a clean, purpose-specific API.

You don't need to understand schemas, indexes, or queries to use extensions—they handle that complexity for you.

Semantic Cache

LLM API calls are expensive and slow. If users ask similar questions, you're paying to generate similar answers repeatedly. Semantic caching solves this by storing responses and returning cached answers when similar prompts are seen again.

How It Works

When a prompt arrives, the cache embeds it and searches for similar cached prompts. If a match is found within the configured distance threshold, the cached response is returned immediately—no LLM call needed. If no match is found, you call the LLM, store the prompt-response pair, and return the response.

The key insight is “similar” rather than “identical.” Traditional caching requires exact matches. Semantic caching matches by meaning, so “What's the capital of France?” and “Tell me France's capital city” can hit the same cache entry.

Threshold Tuning

The distance threshold controls how similar prompts must be to match. A strict threshold (low value, like 0.05) requires near-identical prompts. A loose threshold (higher value, like 0.3) matches more liberally.

Too strict, and you miss valid cache hits. Too loose, and you return wrong answers for different questions. Start strict, monitor cache quality in production, and loosen gradually based on observed behavior.

Multi-Tenant Isolation

In applications serving multiple users or contexts, you often want separate cache spaces. Filters let you scope cache lookups—for example, caching per-user or per-conversation so one user's cached answers don't leak to another.

Redis vs LangCache managed service

SemanticCache stores data in your Redis deployment and uses RedisVL's search index under the hood—you control sizing, networking, and advanced filtering with *FilterExpression*.

If you prefer a hosted semantic cache that is operated as a service you can use `LangCacheSemanticCache` (install `redisvl[langcache]`). It uses the LangCache API endpoint instead of Redis directly. While these are similar, they do not share all the same properties. Refer to *Cache LLM Responses* to see SemanticCache in detail, and *Use LangCache as the LLM Cache Backend* covers `LangCacheSemanticCache` in detail.

Embeddings Cache

Embedding APIs have per-token costs, and computing the same embedding repeatedly wastes money. The embeddings cache stores computed embeddings and returns them on subsequent requests for the same content.

How It Works

Unlike semantic cache (which uses similarity search), embeddings cache uses exact key matching. A deterministic hash is computed from the input text and model name. If that hash exists in the cache, the stored embedding is returned. If not, the embedding is computed, stored, and returned.

This is useful when the same content is embedded multiple times—common in applications where users submit similar queries, or where documents are re-processed periodically.

Wrapping Vectorizers

The embeddings cache can wrap any *vectorizer*, adding transparent caching. Calling the wrapped vectorizer checks the cache first. This requires no changes to your embedding code—just wrap the vectorizer and caching happens automatically.

Message History

LLMs are stateless. To have a conversation, you must include previous messages in each prompt. Message history manages this context, storing conversation turns and retrieving them when building prompts.

Note

`SessionManager` and `SemanticSessionManager` have been renamed to `MessageHistory` and `SemanticMessageHistory`. The old names are deprecated and will be removed in a future release.

Storage Model

Each message includes a role (user, assistant, system, or tool), the message content, a timestamp, and a session identifier. The session tag groups messages into conversations—you might have one session per user, per chat thread, or per agent instance.

Retrieval Strategies

The simplest retrieval is by recency: get the N most recent messages. This works for short conversations but breaks down when context exceeds the LLM's token limit or when relevant information appeared earlier in a long conversation.

Semantic message history adds vector search. Messages are embedded, and you can retrieve by relevance rather than recency. This is powerful for long conversations where the user might reference something said much earlier, or for agents that need to recall specific instructions from their setup.

Session Isolation

Session tags are critical for multi-user applications. Each user’s conversation should be isolated, so retrieving context for User A doesn’t include messages from User B. The session tag provides this isolation, and you can structure sessions however makes sense—per-user, per-thread, per-agent, or any other grouping.

Learn more: *Manage LLM Message History* explains conversation management in detail.

Semantic Router

Semantic routing classifies queries into predefined categories based on meaning. It’s a lightweight alternative to classification models, useful for intent detection, topic routing, and guardrails.

How It Works

You define routes, each with a name and a set of reference phrases that represent that category. The router embeds all references and indexes them. At runtime, an incoming query is embedded and compared against all route references. The route whose references are closest to the query wins—if it’s within the configured distance threshold.

For example, a customer support router might have routes for “billing,” “technical support,” and “account management,” each with 5-10 reference phrases. When a user asks “I can’t log into my account,” the router matches it to the “account management” route based on semantic similarity to that route’s references.

Threshold and Aggregation

Each route has its own distance threshold, controlling how close queries must be to match. Routes can also specify how to aggregate distances when multiple references match—taking the average or minimum distance.

If no route matches (all distances exceed their thresholds), the router returns no match. This lets you handle out-of-scope queries gracefully rather than forcing a classification.

Use Cases

Semantic routing is useful for intent classification (determining what a user wants), topic detection (categorizing content), guardrails (detecting and blocking certain query types), and agent dispatch (sending queries to specialized sub-agents).

Learn more: *Route Queries with SemanticRouter* walks through routing setup in detail.

Related concepts: *Query Types* explains the query types used internally by extensions. *Utilities* covers vectorizers used for embedding.

1.2 Guides

Welcome to the RedisVL guides! Whether you're just getting started or building advanced AI applications, these guides will help you make the most of Redis as your vector database.

Installation **Set up RedisVL.** Install the library and configure your Redis instance for vector search.

[pip install](#) • [Redis Cloud](#) • [Docker](#)

Getting Started **New to RedisVL?** Start here to learn the basics and build your first vector search application in minutes.

[Schema](#) → [Index](#) → [Load](#) → [Query](#)

How-To Guides **Solve specific problems.** Task-oriented recipes for LLM extensions, querying, embeddings, optimization, and storage.

[LLM Caching](#) • [Filtering](#) • [MCP](#) • [Reranking](#) • [Migrations](#)

MCP Setup **Expose Redis through MCP.** Run the RedisVL MCP server, configure one existing index, and use search or optional upsert tools.

[stdio](#), [HTTP](#), [SSE](#) • [One index](#) • [Search and upsert](#)

CLI Reference **Command-line tools.** Manage indices, inspect stats, and work with schemas using the `rvl` CLI.

[rvl index](#) • [rvl stats](#) • [rvl migrate](#) • [Schema YAML](#)

Use Cases **Apply RedisVL to real-world problems.** See which guides map to your use case.

[Agent Context](#) • [Agent Optimization](#) • [Search](#) • [RecSys](#)

1.2.1 Install RedisVL

There are a few ways to install RedisVL. The easiest way is to use `pip`.

Install RedisVL with Pip

Install `redisvl` into your Python (≥ 3.10) environment using `pip`:

```
$ pip install -U redisvl
```

RedisVL comes with a few dependencies that are automatically installed, however, several optional dependencies can be installed separately based on your needs:

```
# Vectorizer providers
$ pip install redisvl[openai]           # OpenAI embeddings
$ pip install redisvl[cohere]          # Cohere embeddings and reranking
$ pip install redisvl[mistralai]       # Mistral AI embeddings
$ pip install redisvl[voyageai]        # Voyage AI embeddings and reranking
$ pip install redisvl[sentence-transformers] # HuggingFace local embeddings
$ pip install redisvl[google-genai]    # Google embeddings (Vertex AI + Gemini API)
$ pip install redisvl[vertexai]        # Google Vertex AI embeddings (legacy; ⚠
↪ deprecated, multimodal only)
$ pip install redisvl[bedrock]         # AWS Bedrock embeddings
```

(continues on next page)

(continued from previous page)

```
# Other optional features
$ pip install redisvl[mcp]           # RedisVL MCP server support (Python 3.10+)
$ pip install redisvl[langcache]    # LangCache managed service integration
$ pip install redisvl[sql-redis]    # SQL query support
```

If you use ZSH, remember to escape the brackets:

```
$ pip install redisvl\[openai\]
```

You can install multiple optional dependencies at once:

```
$ pip install redisvl[mcp,openai,cohere,sentence-transformers]
```

To install **all** optional dependencies at once:

```
$ pip install redisvl[all]
```

Install RedisVL from Source

To install RedisVL from source, clone the repository and install the package using pip:

```
$ git clone https://github.com/redis/redis-vl-python.git && cd redis-vl-python
$ pip install .

# or for an editable installation (for developers of RedisVL)
$ pip install -e .
```

Development Installation

For contributors who want to develop RedisVL, we recommend using **uv** for dependency management:

```
# Clone the repository
$ git clone https://github.com/redis/redis-vl-python.git && cd redis-vl-python

# Install uv if you don't have it
$ pip install uv

# Install all dependencies (including dev and docs)
$ uv sync

# Or use make
$ make install
```

This installs the package in editable mode along with all development dependencies (testing, linting, type checking) and documentation dependencies.

Running Tests and Linting

```
# Run tests (no external APIs required)
$ make test

# Run all tests (includes API-dependent tests)
$ make test-all

# Format code
$ make format

# Run type checking
$ make check-types

# Run full check (lint + test)
$ make check
```

Pre-commit Hooks

We use pre-commit hooks to ensure code quality. Install them with:

```
$ pre-commit install
```

Run hooks manually on all files:

```
$ pre-commit run --all-files
```

Installing Redis

RedisVL requires Redis with [Redis Search](#) available. There are several options:

1. [Redis Cloud](#), a fully managed cloud offering with a free tier
2. [Redis 8+ \(Docker\)](#), for local development and testing
3. [Redis Enterprise](#), a commercial self-hosted option

Redis Cloud

Redis Cloud is the easiest way to get started with RedisVL. You can sign up for a free account [here](#). Make sure to have [Redis Search](#) enabled when creating your database.

Redis 8+ (local development)

For local development and testing, we recommend running Redis 8+ in a Docker container:

```
docker run -d --name redis -p 6379:6379 redis:8.4
```

Redis 8 includes Redis Search and built-in vector search capabilities.

Redis Enterprise (self-hosted)

Redis Enterprise is a commercial offering that can be self-hosted. You can download the latest version [here](#).

If you are considering a self-hosted Redis Enterprise deployment on Kubernetes, there is the [Redis Enterprise Operator](#) for Kubernetes. This will allow you to easily deploy and manage a Redis Enterprise cluster on Kubernetes.

Redis Sentinel

For high availability deployments, RedisVL supports connecting to Redis through Sentinel. Use the `redis+sentinel://` URL scheme to connect. Both sync and async connections are fully supported.

```
from redisvl.index import SearchIndex, AsyncSearchIndex

# Sync connection via Sentinel
# Format: redis+sentinel://[username:password@]host1:port1,host2:port2/service_name[/db]
index = SearchIndex.from_yaml(
    "schema.yaml",
    redis_url="redis+sentinel://sentinel1:26379,sentinel2:26379/mymaster"
)

# Async connection via Sentinel
async_index = AsyncSearchIndex.from_yaml(
    "schema.yaml",
    redis_url="redis+sentinel://sentinel1:26379,sentinel2:26379/mymaster"
)

# With authentication and database selection
index = SearchIndex.from_yaml(
    "schema.yaml",
    redis_url="redis+sentinel://user:pass@sentinel1:26379,sentinel2:26379/mymaster/0"
)
```

The Sentinel URL format supports:

- Multiple sentinel hosts (comma-separated)
- Optional authentication (username:password)
- Service name (defaults to `mymaster` if not specified)
- Optional database number (defaults to 0)
- Both sync (`SearchIndex`) and async (`AsyncSearchIndex`) connections

Redis permissions (ACLs)

RedisVL reaches Redis through Redis Search commands, but not all of them need the `@search` category. Querying and loading work under an ordinary `+@read +@write` role; it is the commands that inspect or manage an index — `FT.INFO`, `FT.CREATE`, `FT._LIST` — that need `@search` or an explicit grant.

The command-to-category mapping below was measured against live servers rather than quoted from published documentation, which does not list ACL categories per `FT.*` command. It was identical on Redis 8.0.6, 8.2.7, 8.4.5, 8.6.4, 8.8.0 and 8.8.1. Check your own deployment with `COMMAND INFO ft.info`, `ACL CAT search`, or `ACL DRYRUN <user> FT.INFO <index>`.

What each operation needs

Operation	Redis command	+@all -@admin	+@read +@write
<code>index.query()</code> , <code>index.search()</code> , <code>index.aggregate()</code>	<code>FT.SEARCH</code> , <code>FT.AGGREGATE</code>	Yes	Yes
<code>index.load()</code>	<code>HSET</code> or <code>JSON.SET</code> (needs key access)	Yes	Yes
<code>index.clear()</code>	<code>FT.SEARCH</code> , then <code>DEL</code> per batch	Yes	Yes
<code>index.exists()</code> , <code>index.info()</code> , <code>SearchIndex.from_existing()</code> , <code>rvl index info</code> , <code>rvl stats</code>	<code>FT.INFO</code>	Yes	No
<code>index.create()</code>	<code>FT.CREATE</code>	Yes	No
<code>index.delete()</code> , <code>rvl index delete</code> , <code>rvl index destroy</code>	<code>FT.DROPINDEX</code>	Yes	Yes
Enumerating indexes (see below)	<code>FT._LIST</code>	No	No

Except for `FT.CREATE`, every Yes above assumes key patterns that cover the index prefix — see [Key permissions](#). `FT.CREATE` is not checked against those patterns, so a credential can create an index it cannot query. Adding `-@dangerous` to the second column additionally denies `FT.DROPINDEX`, so `index.delete()` becomes No. An SVS-VAMANA schema needs more than `FT.CREATE`: `index.create()` first probes capabilities with `INFO (@slow @dangerous)` and `MODULE LIST (@admin @slow @dangerous)`, so both `-@admin` and `-@dangerous` policies break creation for those schemas.

Two of the rows above deserve their own explanation.

`FT._LIST` is tagged `@admin` as well as `@search` and `@slow`, and ACL rules are applied left to right — so a rule that grants search access and then takes back administrative commands, such as `+@search -@admin` or `+@all -@admin`, denies it:

User <name> has no permissions to run the 'FT._LIST' command

`FT._LIST` does not *require* `@admin`: granting `+@search` on its own permits it. Only rules that subtract `@admin` after granting search are affected. To keep such a policy and still enumerate indexes, grant the command back explicitly with `+ft._list`. Enumeration is reached by `SearchIndex.listall()` and `AsyncSearchIndex.listall()`, by `rvl index listall`, and by the migration entry points that discover indexes for you: `rvl migrate helper`, `rvl migrate wizard` when no `-i/--index` is given, and `rvl migrate batch-plan --pattern`.

`FT.DROPINDEX` is tagged `@dangerous` and `@write` as well as `@search`, so a policy that subtracts `@dangerous` denies `index.delete()`, `rvl index delete`, and `rvl index destroy`.

Roles built from @read and @write

FT.INFO is in neither @read nor @write — its only category is @search — and FT.CREATE is the same. So an application role assembled from +@read +@write -@dangerous — a natural least-privilege shape for a runtime that must query and load but must not manage indexes — can query and load, but cannot ask whether an index exists and cannot create or drop one. Subtracting @dangerous is not what denies FT.INFO or FT.CREATE: +@all -@dangerous permits both. Those commands are simply never granted by @read or @write.

Every extension constructor checks whether its index exists, so under such a credential all of them fail while being constructed:

```
RedisSearchError: Error while fetching llmcache index info:
User <name> has no permissions to run the 'FT.INFO' command
```

Reading a cache is not read-only either. When a cache has a TTL configured — at construction or later via `set_ttl()` — every read that hits refreshes the matched entries' TTL, so it issues EXPIRE. That includes `SemanticCache.check()`, every `EmbeddingsCache` getter, their async equivalents, and the reads a vectorizer performs for you when constructed with `cache=`. EXPIRE is in @write, not @read (measured with ACL `CAT write`), so a lookup-only credential fails on a cache *hit* rather than on the write that populated it.

Granting the command is only half of it, because the key patterns must permit writes too. Measured on 8.4.5 with ACL `DRYRUN <user> EXPIRE llmcache:abc 60`:

Rules	Result
+@read ~llmcache:*	no permissions to run the 'expire' command
+@read +expire ~llmcache:*	Permitted
+@read +expire %R~llmcache:*	no permissions to access the 'llmcache:abc' key

So `%R~llmcache:*` — the read-only shape the [Key permissions](#) table below presents as sufficient for querying — is denied on the key, not the command. +@read +@write is unaffected, as is a cache with no TTL. See [Cache LLM Responses](#) for the refresh behaviour itself, including that `set_ttl()` starts adding TTLs to entries stored without one.

“no permissions to run the ‘FT.INFO’ command”

RedisVL does not guess its way around this. A credential that cannot ask whether the index exists also cannot create one, so there is nothing useful to infer — instead, tell RedisVL that the index is already there:

```
cache = SemanticCache(
    name="llmcache",
    redis_url="redis://localhost:6379",
    create_index=False,
)
```

`create_index=False` is available on `SemanticCache`, `MessageHistory`, `SemanticMessageHistory` and `SemanticRouter`. It skips the existence check, the comparison of your schema against the live index, and index creation — the constructor issues no index command at all. Pass it when the index is managed externally, or when the credential cannot run FT.INFO. It cannot be combined with `overwrite=True`, which asks for the opposite.

A `SearchIndex` used directly needs nothing special: build it with `from_dict()` or `from_yaml()`, then load and query. The methods that stay unavailable are the ones that read index metadata — `exists()`, `info()`, and `from_existing()`, which reconstructs a schema out of Redis. `clear()` is not among them: it enumerates with FT.SEARCH and deletes in batches, so it needs no more than querying does.

The flag also skips the SVS-VAMANA capability probe described above, since that runs inside `create()`.

Provisioning a router

`SemanticRouter` with `create_index=False` writes nothing at all: not the reference vectors for its routes, and not the stored route config that `SemanticRouter.from_existing()` reads. Preparing a router for this mode therefore means constructing it once with a privileged credential — a hand-written `FT.CREATE` is not enough, because the reference vectors have to be embedded and written too. Without them the router matches nothing, which looks like a distance-threshold problem rather than an empty index.

Afterwards, `SemanticRouter.from_existing(name, create_index=False)` is the way to attach to it: it recovers the routes and thresholds with `JSON.GET` and needs no `FT.INFO`. The stored config must contain the full route set. Each route's distance threshold is applied from that recovered list, so an incomplete stored config silently narrows matching — and `add_route()` and `remove_route()` rewrite the stored config from the same list, so mutating an incomplete config permanently drops the omitted routes from the config every other client reads.

When the schema diverges

With `create_index=False` nothing verifies that the live index matches the schema you described. Some mismatches are loud on first use, and several are silent:

Mismatch	What happens
The index does not exist	<code>RedisSearchError</code> on the first query, naming the missing index
Vector dimensions disagree	Error parsing vector similarity query: query vector blob size (32) does not match index's expected size (16) — but only once the index holds a document. On an empty index the same query returns nothing, so a freshly provisioned index hides this until the first write lands
The prefix does not cover your keys	Silent. Documents are written but never indexed, so queries return nothing, forever
The index is ON JSON and you write hashes (or the reverse)	Silent , the same way
The datatype or distance metric differs	Silent. Neither is restated by a query, so nothing compares them — results come back ranked by the index's metric, not yours

For the silent cases the tell is `FT.INFO`'s `key_type`, `prefixes` and `attributes` — not `hash_indexing_failures`, which stays `0` because those keys were never indexing candidates. Diagnosing it therefore needs a credential that can run `FT.INFO`.

What an attach-only instance may still do

Removing *entries* is available on every path, and is how a caller invalidates an externally managed cache without holding the provisioning credential: `clear()` (plus `SemanticCache.aclear()`), and targeted removal of a specific cache entry, message or route. None of it removes the index, and all of it runs under `+@read +@write`.

What `create_index=False` refuses is `delete()` (and `SemanticCache.delete()`), because that drops the index. Refusing it protects an externally managed index — including one reached through an alias — from being destroyed through an attach-only instance. Drop the index through the privileged provisioning path that owns it.

The two kinds of `clear()` decide *which keys go* differently, and neither choice is verified against the live index under this flag:

Method	Deletes	Chooses keys by
<code>SemanticCache.clear()</code> , <code>aclear()</code>	every key under <code>{name}:</code>	SCAN/DEL on the prefix this instance declares — no index command at all
<code>MessageHistory.clear()</code> , <code>SemanticMessageHistory.clear()</code> , <code>SemanticRouter.clear()</code>	every document the live index covers	FT.SEARCH paging via <code>SearchIndex.clear()</code>

FT.SEARCH is in @read as well as @search, so a +@read +@write credential is granted it — unlike FT.INFO, which is in neither and is what made these three unavailable before. Note that FT.SEARCH additionally requires the credential's key patterns to be a superset of the index prefixes, the same rule described under [Key permissions](#).

Because the two enumerate differently, they fail differently, and the section above is what decides which failure you get. Both are silent:

- **Prefix-based clearing deletes too much, or nothing.** SCAN/DEL is blind to the index and to the key type, so it removes every key under `{name}:` — another writer's entries, and unrelated application data sharing that namespace root. And if the live index covers a *different* prefix, or is an alias onto one, `clear()` deletes only what this instance itself wrote and leaves every served entry in place: it reports success and the cache still returns the stale hits you called it to invalidate.
- **Index-based clearing deletes documents you never wrote.** `SearchIndex.clear()` deletes what the live index covers, so against an index on a different prefix — or a multi-PREFIX index, or an alias — it removes another application's documents while leaving this instance's own unindexed entries behind.

Diagnosing either needs FT.INFO, which is the command an attach-only credential does not have. If the index is provisioned for you, get its `prefixes` and `key_type` from whoever provisions it and make your extension's name match, rather than inferring it from a successful query.

Key permissions

Command categories are only half of it. Redis also scopes the search commands by key pattern: [the ACL documentation](#) states that only users with access to a *superset* of the prefixes defined at index creation can create, modify, or read an index.

Measured on 8.4.5 against an index prefixed `doc:`, with the command categories held constant at +@all:

Key patterns	FT.SEARCH, FT.INFO, FT.AGGREGATE
<code>~doc:*</code> (superset)	Permitted
<code>%R~doc:*</code> (read permission only)	Permitted
<code>~doc:1</code> (partial overlap)	NOPERM User does not have the required permissions to query the index
<code>~other:*</code> (no overlap)	The same denial

Partial overlap is worth emphasising: it fails exactly like no overlap at all, rather than returning the subset you can read. FT.CREATE is not checked this way, so a credential can create an index it is then unable to query.

Key patterns are glob-style, matched by the same engine as SCAN MATCH, so metacharacters in a prefix do not mean what they look like. Measured on 8.4.5, `~cache[ab]:*` grants `cachea:1` and `cacheb:1` while *denying* the literal key

`cache[ab]:1`; escaped as `~cache\[ab]:*` it does the reverse. Escape `*`, `?`, `[` and `\` in a prefix, and confirm the rule with `ACL DRYRUN`.

`create_index=False` does not help here — the very commands it lets you avoid are joined by the ones it cannot, so widen the key patterns instead.

Outside of Redis Search, RedisVL identifies itself on connect with `CLIENT SETINFO`. That command is tagged `@connection` and `@slow`, and belongs to neither `@read` nor `@write`, so a rule built up from those categories never grants it. A credential that cannot run it still connects: identification only populates the `lib-name` field that `CLIENT LIST` and `CLIENT INFO` display, so a refusal is ignored (and logged, if you have configured logging at debug level). Grant `+client|setinfo` if you want RedisVL to appear as the connecting library there — note that this labels the connection RedisVL opens, while `redis-py` labels the rest of the pool as plain `redis-py`.

Cluster deployments need one more grant. `RedisCluster` discovers the topology with `CLUSTER SLOTS`, which is tagged `@slow` only, so a credential assembled from `+@read +@write` cannot open a clustered connection at all — `redis-py` reports this as `Redis Cluster cannot be connected`, with the underlying permission error chained beneath it. Grant `+cluster|slots` alongside the rules above.

Redis Cloud and Redis Software

Both manage ACLs through their own control plane rather than the `ACL SETUSER` command, so permissions are attached to roles and users instead of being set on a connection.

- **Redis Cloud** provides three predefined ACL rules that cannot be edited — Full-Access, Read-Write (“read and write commands and excludes dangerous commands”), and Read-Only — which you assign to a data access role. See [Configure permissions with Redis ACLs](#). Custom rules use the same syntax as above.
- **Redis Software** ships one predefined ACL, Full Access, and you define others in the Cluster Manager UI or with a `POST /v1/redis-acls` request. It [does not support every ACL command](#), nor nested selectors, nor `(` and `)` in key patterns.

Because the predefined rules’ exact command sets are not published, confirm a credential against the database rather than inferring what its policy name implies — `ACL DRYRUN <user> FT.INFO <index>` answers it directly. Read the descriptions carefully before assuming you are unaffected: Read-Only allows read commands, and Read-Write “allows read and write commands and excludes dangerous commands”, so both read as `@read/@write`-shaped — the shape that denies `FT.INFO` and `FT.CREATE` and wants `create_index=False`. Only Full-Access is clearly unaffected.

Redis Software’s documentation uses `+@read +FT.INFO +FT.SEARCH` as an example rule, which is a good illustration: it permits querying and `index.exists()`, but not `index.create()` or index enumeration. Grant `FT.CREATE` explicitly when the application creates its own index. When the index is provisioned for the application instead, leave it out and construct with `create_index=False`.

1.2.2 Getting Started

RedisVL is a Python library with an integrated CLI for building AI applications with Redis. This guide covers the core workflow:

1. Defining an `IndexSchema`
2. Preparing a sample dataset
3. Creating a `SearchIndex`
4. Using the `rvl` CLI
5. Loading data into Redis
6. Fetching and managing records

7. Executing vector searches
8. Updating an index

Prerequisites

Before you begin, ensure you have:

- Installed RedisVL: `pip install redisvl`
- A running Redis instance ([Redis 8+](#) or [Redis Cloud](#))

What You'll Learn

By the end of this guide, you will be able to:

- Create index schemas using Python dictionaries or YAML files
- Build and manage `SearchIndex` objects
- Use the `rvl` CLI for index management
- Load data and execute vector similarity searches
- Fetch individual records and list all keys in an index
- Delete specific records by key or document ID
- Update index schemas as your application evolves

Define an IndexSchema

The `IndexSchema` maintains crucial **index configuration** and **field definitions** to enable search with Redis. For ease of use, the schema can be constructed from a python dictionary or yaml file.

Example Schema Creation

Consider a dataset with user information, including `job`, `age`, `credit_score`, and a 3-dimensional `user_embedding` vector.

You must also decide on a Redis index name and key prefix to use for this dataset. Below are example schema definitions in both YAML and Dict format.

YAML Definition:

```
version: '0.1.0'

index:
  name: user_simple
  prefix: user_simple_docs

fields:
  - name: user
    type: tag
  - name: credit_score
    type: tag
```

(continues on next page)

(continued from previous page)

```

- name: job
  type: text
- name: age
  type: numeric
- name: user_embedding
  type: vector
  attrs:
    algorithm: flat
    dims: 3
    distance_metric: cosine
    datatype: float32

```

Store this in a local file, such as `schema.yaml`, for RedisVL usage.

Python Dictionary:

```

schema = {
  "index": {
    "name": "user_simple",
    "prefix": "user_simple_docs",
  },
  "fields": [
    {"name": "user", "type": "tag"},
    {"name": "credit_score", "type": "tag"},
    {"name": "job", "type": "text"},
    {"name": "age", "type": "numeric"},
    {
      "name": "user_embedding",
      "type": "vector",
      "attrs": {
        "dims": 3,
        "distance_metric": "cosine",
        "algorithm": "flat",
        "datatype": "float32"
      }
    }
  ]
}

```

Sample Dataset Preparation

Below, create a mock dataset with `user`, `job`, `age`, `credit_score`, and `user_embedding` fields. The `user_embedding` vectors are synthetic examples for demonstration purposes.

For more information on creating real-world embeddings, refer to this [article](#).

```

import numpy as np

data = [
  {
    'user': 'john',

```

(continues on next page)

(continued from previous page)

```

    'age': 1,
    'job': 'engineer',
    'credit_score': 'high',
    'user_embedding': np.array([0.1, 0.1, 0.5], dtype=np.float32).tobytes()
},
{
    'user': 'mary',
    'age': 2,
    'job': 'doctor',
    'credit_score': 'low',
    'user_embedding': np.array([0.1, 0.1, 0.5], dtype=np.float32).tobytes()
},
{
    'user': 'joe',
    'age': 3,
    'job': 'dentist',
    'credit_score': 'medium',
    'user_embedding': np.array([0.9, 0.9, 0.1], dtype=np.float32).tobytes()
}
]

```

The `user_embedding` vectors are converted to bytes using NumPy's `.tobytes()` method.

Create a SearchIndex

With the schema and sample dataset ready, create a `SearchIndex`.

Bring your own Redis connection instance

This is ideal in scenarios where you have custom settings on the connection instance or if your application will share a connection pool:

```

from redisvl.index import SearchIndex
from redis import Redis

client = Redis.from_url("redis://localhost:6379")
index = SearchIndex.from_dict(schema, redis_client=client, validate_on_load=True)

```

Let the index manage the connection instance

This is ideal for simple cases:

```

index = SearchIndex.from_dict(schema, redis_url="redis://localhost:6379", validate_on_
↪load=True)

# If you don't specify a client or Redis URL, the index will attempt to
# connect to Redis at the default address "redis://localhost:6379".

```

Create the index

Now that we are connected to Redis, we need to run the create command.

```
index.create(overwrite=True)
```

Note that at this point, the index has no entries. Data loading follows.

Inspect with the rv1 CLI

Use the rv1 CLI to inspect the created index and its fields:

```
!rv1 index info -i user_simple
```

Index Information:					
Index Name	Storage Type	Prefixes	Index Options		
↪ Indexing					
user_simple	HASH	['user_simple_docs']	[]		
↪ 0					
Index Fields:					
Name	Attribute	Type	Field Option	Option Value	
↪ Field Option	Option Value	Field Option	Option Value	Field	
↪ Option	Option Value				
user	user	TAG	SEPARATOR	,	
↪					
↪					
credit_score	credit_score	TAG	SEPARATOR	,	
↪					
↪					
job	job	TEXT	WEIGHT	1	
↪					
↪					
age	age	NUMERIC			
↪					
↪					
user_embedding	user_embedding	VECTOR	algorithm	FLAT	
↪ data_type	FLOAT32	dim	3	distance_	
↪ metric	COSINE				

Load Data to SearchIndex

Load the sample dataset to Redis.

Validate data entries on load

RedisVL uses pydantic validation under the hood to ensure loaded data is valid and confirms to your schema. This setting is optional and can be configured in the `SearchIndex` class.

```
keys = index.load(data)

print(keys)
```

```
['user_simple_docs:01KHKHQYX95EDQN18FG8FRMRQ5', 'user_simple_
docs:01KHKHQYXC97WY4ACG1V01GEPC', 'user_simple_docs:01KHKHQYXC97WY4ACG1V01GEPD']
```

By default, load will create a unique Redis key as a combination of the index key `prefix` and a random ULID. You can also customize the key by providing direct keys or pointing to a specified `id_field` on load.

Load INVALID data

This will raise a `SchemaValidationError` if `validate_on_load` is set to true in the `SearchIndex` class.

```
# NBVAL_SKIP

try:
    keys = index.load([{"user_embedding": True}])
except Exception as e:
    print(str(e))
```

```
Schema validation failed for object at index 0. Field 'user_embedding' expects bytes
(vector data), but got boolean value 'True'. If this should be a vector field, provide
a list of numbers or bytes. If this should be a different field type, check your
schema definition.
```

```
Object data: {
  "user_embedding": true
}
```

```
Hint: Check that your data types match the schema field definitions. Use index.schema.
fields to view expected field types.
```

Upsert the index with new data

Upsert data by using the load method again:

```
# Add more data
new_data = [{
    'user': 'tyler',
    'age': 9,
    'job': 'engineer',
```

(continues on next page)

(continued from previous page)

```
'credit_score': 'high',
'user_embedding': np.array([0.1, 0.3, 0.5], dtype=np.float32).tobytes()
}]
keys = index.load(new_data)

print(keys)

['user_simple_docs:01KHKHR37CD6143DNQ41G3ADNA']
```

Fetch and Manage Records

RedisVL provides several methods to retrieve and manage individual records in your index.

Fetch a record by ID

Use `fetch()` to retrieve a single record when you know its ID. The ID is the unique identifier you provided during load (via `id_field`) or the auto-generated ULID.

```
# Fetch a record by its ID (e.g., the user field value if used as id_field)
# First, let's reload data with a specific id_field
index.load(data, id_field="user")

# Now fetch by the user ID
record = index.fetch("john")
print(record)
```

You can also construct the full Redis key from an ID using the `key()` method:

```
# Get the full Redis key for a given ID
full_key = index.key("john")
print(f"Full Redis key: {full_key}")
```

```
<IPython.core.display.HTML object>
```

List all keys in the index

To enumerate all keys in your index, use `paginate()` with a `FilterQuery`. This is useful for batch processing or auditing your data.

```
from redisvl.query import FilterQuery
from redisvl.query.filter import FilterExpression

# Create a query that matches all documents
query = FilterQuery(
    filter_expression=FilterExpression(""),
    return_fields=["user", "age", "job"]
)
```

(continues on next page)

(continued from previous page)

```
# Paginate through all results
for batch in index.paginate(query, page_size=10):
    for doc in batch:
        print(f"Key: {doc['id']}, User: {doc['user']}")
```

Delete specific records

Use `drop_keys()` to remove specific records by their full Redis key, or `drop_documents()` to remove by document ID.

```
# Delete by full Redis key
full_key = index.key("john")
deleted_count = index.drop_keys(full_key)
print(f"Deleted {deleted_count} record(s) by key")

# Delete multiple keys at once
# index.drop_keys(["key1", "key2", "key3"])
```

```
# Delete by document ID (without the prefix)
deleted_count = index.drop_documents("mary")
print(f"Deleted {deleted_count} record(s) by document ID")

# Delete multiple documents at once
# index.drop_documents(["id1", "id2", "id3"])
```

Note: `drop_keys()` expects the full Redis key (including prefix), while `drop_documents()` expects just the document ID.

Creating VectorQuery Objects

Next we will create a vector query object for our newly populated index. This example will use a simple vector to demonstrate how vector similarity works. Vectors in production will likely be much larger than 3 floats and often require Machine Learning models (i.e. Huggingface sentence transformers) or an embeddings API (Cohere, OpenAI). `redisvl` provides a set of `Vectorizers` to assist in vector creation.

```
from redisvl.query import VectorQuery
from jupyterutils import result_print

query = VectorQuery(
    vector=[0.1, 0.1, 0.5],
    vector_field_name="user_embedding",
    return_fields=["user", "age", "job", "credit_score", "vector_distance"],
    num_results=3
)
```

```
<IPython.core.display.HTML object>
```

Note: For HNSW and SVS-VAMANA indexes, you can tune search performance using runtime parameters:


```
# Example with HNSW runtime parameters
query = VectorQuery(
    vector=[0.1, 0.1, 0.5],
    vector_field_name="user_embedding",
    return_fields=["user", "age", "job"],
    num_results=3,
    ef_runtime=50 # Higher for better recall (HNSW only)
)
```

See the *SVS-VAMANA guide* and *Advanced Queries guide* for more details on runtime parameters.

Executing queries

With our `VectorQuery` object defined above, we can execute the query over the `SearchIndex` using the `query` method.

```
results = index.query(query)
result_print(results)
```

Using an Asynchronous Redis Client

The `AsyncSearchIndex` class along with an async Redis python client allows for queries, index creation, and data loading to be done asynchronously. This is the recommended route for working with `redisvl` in production-like settings.

```
from redisvl.index import AsyncSearchIndex
from redis.asyncio import Redis

client = Redis.from_url("redis://localhost:6379")
index = AsyncSearchIndex.from_dict(schema, redis_client=client)
```

```
4
```

```
# execute the vector query async
results = await index.query(query)
result_print(results)
```

```
True
```

Updating a schema

In some scenarios, it makes sense to update the index schema. With Redis and `redisvl`, this is easy because Redis can keep the underlying data in place while you change or make updates to the index configuration.

So for our scenario, let's imagine we want to reindex this data in 2 ways:

- by using a `Tag` type for `job` field instead of `Text`
- by using an `hnsf` vector index for the `user_embedding` field instead of a `flat` vector index

```
# Modify this schema to have what we want
index.schema.remove_field("job")
index.schema.remove_field("user_embedding")
index.schema.add_fields([
    {"name": "job", "type": "tag"},
    {
        "name": "user_embedding",
        "type": "vector",
        "attrs": {
            "dims": 3,
            "distance_metric": "cosine",
            "algorithm": "hnsw",
            "datatype": "float32"
        }
    }
])
```

```
# Run the index update but keep underlying data in place
await index.create(overwrite=True, drop=False)
```

```
# Execute the vector query async
results = await index.query(query)
result_print(results)
```

Check Index Stats

Use the `rvl` CLI to check the stats for the index:

```
!rvl stats -i user_simple
```

Next Steps

Now that you understand the basics of RedisVL, explore these related guides:

- [Query and Filter Data](#) - Learn advanced filtering with tag, numeric, text, and geo filters
- [Create Embeddings with Vectorizers](#) - Generate embeddings using OpenAI, HuggingFace, Cohere, and more
- [Choose a Storage Type](#) - Understand when to use Hash vs JSON storage

Cleanup

Use `.clear()` to flush all data from Redis associated with the index while leaving the index in place for future insertions.

Use `.delete()` to remove both the index and the underlying data.

```
# Clear all data from Redis associated with the index
await index.clear()
```

```
# But the index is still in place
await index.exists()
```

```
# Remove / delete the index in its entirety
await index.delete()
```

1.2.3 How-To Guides

How-to guides are **task-oriented** recipes that help you accomplish specific goals. Each guide focuses on solving a particular problem and can be completed independently.

LLM Extensions

- *Cache LLM Responses*: semantic caching to reduce costs and latency
- *Use LangCache as the LLM cache*: managed cache service with LangCache
- *Manage LLM Message History*: persistent chat history with relevancy retrieval
- *Route Queries with SemanticRouter*: classify intents and route queries

Querying

- *Query and Filter Data*: combine tag, numeric, geo, and text filters
- *Use Advanced Query Types*: hybrid, multi-vector, range, and text queries
- *Write SQL Queries for Redis*: translate SQL to Redis query syntax

Embeddings

- *Create Embeddings with Vectorizers*: OpenAI, Cohere, HuggingFace, and more
- *Cache Embeddings*: reduce costs by caching embedding vectors

Optimization

- *Rerank Search Results*: improve relevance with cross-encoders and rerankers
- *Optimize Indexes with SVS-VAMANA*: graph-based vector search with compression

Storage

- *Choose a Storage Type*: Hash vs JSON formats and nested data
- *Migrate an Index*: use the migrator helper, wizard, plan, apply, and validate workflow
- *Migrate an Index: Quantization, Resume, Backup, Wizard*: hands-on notebook for vector quantization with crash-safe resume, rollback, and wizard flow

CLI Operations

- *Manage Indices with the CLI*: create, inspect, and delete indices from your terminal
- *Run RedisVL MCP*: expose an existing Redis index to MCP clients
- *Authenticate RedisVL MCP*: require JWT bearer tokens and gate read vs write

Quick Reference

I want to...	Guide
Cache LLM responses	Cache LLM Responses
Use LangCache (managed) for LLM caching	Use LangCache as the LLM cache
Store chat history	Manage LLM Message History
Route queries by intent	Route Queries with SemanticRouter
Filter results by multiple criteria	Query and Filter Data
Use hybrid or multi-vector queries	Use Advanced Query Types
Translate SQL to Redis	Write SQL Queries for Redis
Choose an embedding model	Create Embeddings with Vectorizers
Speed up embedding generation	Cache Embeddings
Improve search accuracy	Rerank Search Results
Optimize index performance	Optimize Indexes with SVS-VAMANA
Decide on storage format	Choose a Storage Type
Manage indices from terminal	Manage Indices with the CLI
Expose an index through MCP	Run RedisVL MCP
Authenticate the MCP server	Authenticate RedisVL MCP
Plan and run a supported index migration	Migrate an Index
Quantize vectors with resume, rollback, and the wizard	Migrate an Index: Quantization, Resume, Backup, Wizard

Cache LLM Responses

This guide demonstrates how to use RedisVL's `SemanticCache` to cache LLM responses based on semantic similarity. Semantic caching reduces API costs and latency by retrieving cached responses for semantically similar prompts instead of making redundant API calls.

Prerequisites

Before you begin, ensure you have:

- Installed RedisVL: `pip install redisvl`
- A running Redis instance ([Redis 8+](#) or [Redis Cloud](#))
- An OpenAI API key for the examples

What You'll Learn

By the end of this guide, you will be able to:

- Set up and configure a `SemanticCache`
- Store and retrieve cached LLM responses
- Understand entry IDs and keys for fetching and deleting specific entries
- Customize semantic similarity thresholds
- Configure TTL policies and understand TTL refresh behavior
- Implement access controls with tags and filters for multi-user scenarios

First, import `OpenAI` to use their API for responding to user prompts. The following code creates a simple `ask_openai` helper method to assist.

```
import os
import getpass
import time
import numpy as np

from openai import OpenAI

os.environ["TOKENIZERS_PARALLELISM"] = "False"

api_key = os.getenv("OPENAI_API_KEY") or getpass.getpass("Enter your OpenAI API key: ")

client = OpenAI(api_key=api_key)

def ask_openai(question: str) -> str:
    response = client.completions.create(
        model="gpt-4o-mini",
        prompt=f"Answer the following question simply: {question}",
        max_tokens=200
    )
    return response.choices[0].text.strip()
```

```
# Test
print(ask_openai("What is the capital of France?"))
```

```
Paris.
```

Initializing SemanticCache

`SemanticCache` will automatically create an index within Redis upon initialization for the semantic cache content.

```
import warnings
warnings.filterwarnings('ignore')

from redisvl.extensions.cache.llm import SemanticCache
from redisvl.utils.vectorize import HFTextVectorizer

llmcache = SemanticCache(
    name="llmcache", # underlying search index
    redis_url="redis://localhost:6379", # redis connection url
    distance_threshold=0.1, # semantic cache distance
    vectorizer=HFTextVectorizer("redis/langcache-embed-v2"), # embedding model
)
```

You try to use a model that was created with version 4.1.0, however, your version is 3.4.
→ 1. This might cause unexpected behavior or errors. In that case, try to update to the
→ latest version.

```
# look at the index specification created for the semantic cache lookup
!rvl index info -i llmcache
```

Index Information:

Index Name	Storage Type	Prefixes	Index Options	Indexing	
llmcache	HASH	['llmcache']	[]	0	

Index Fields:

Name	Attribute	Type	Field Option	Option Value	
→ Field Option	→ Option Value	→ Field Option	→ Option Value	→ Field	
→ Option	→ Option Value				
prompt	prompt	TEXT	WEIGHT	1	
→	→	→	→	→	
→	→	→	→	→	
response	response	TEXT	WEIGHT	1	
→	→	→	→	→	
→	→	→	→	→	
inserted_at	inserted_at	NUMERIC			
→	→	→	→	→	
→	→	→	→	→	
updated_at	updated_at	NUMERIC			
→	→	→	→	→	
→	→	→	→	→	
prompt_vector	prompt_vector	VECTOR	algorithm	FLAT	
→ data_type	→ FLOAT32	→ dim	→ 768	→ distance_	
→ metric	→ COSINE				

Basic Cache Usage

```
question = "What is the capital of France?"
```

```
# Check the semantic cache -- should be empty
if response := llmcache.check(prompt=question):
    print(response)
else:
    print("Empty cache")
```

Empty cache

Our initial cache check should be empty since we have not yet stored anything in the cache. Below, store the question, proper response, and any arbitrary metadata (as a python dictionary object) in the cache.

```
# Cache the question, answer, and arbitrary metadata
llmcache.store(
    prompt=question,
    response="Paris",
    metadata={"city": "Paris", "country": "france"}
)
```

```
'llmcache:115049a298532be2f181edb03f766770c0db84c22aff39003fec340deaec7545'
```

Now check the cache again with the same question and with a semantically similar question:

```
# Check the cache again
if response := llmcache.check(prompt=question, return_fields=["prompt", "response",
↪ "metadata"]):
    print(response)
else:
    print("Empty cache")
```

```
[{'prompt': 'What is the capital of France?', 'response': 'Paris', 'metadata': {'city':
↪ 'Paris', 'country': 'france'}, 'key':
↪ 'llmcache:115049a298532be2f181edb03f766770c0db84c22aff39003fec340deaec7545'}]
```

```
# Check for a semantically similar result
question = "What actually is the capital of France?"
llmcache.check(prompt=question)[0]['response']
```

```
'Paris'
```

Entry IDs and Keys

Each cache entry has two identifiers:

- **entry_id**: A deterministic hash of the prompt + filters. Used to identify the entry within the cache.
- **key**: The full Redis key (prefix + entry_id). Used for direct Redis operations.

The entry_id is generated as a SHA256 hash of the prompt and any filters, meaning:

- Same prompt + same filters = same entry_id (overwrites previous entry)
- Same prompt + different filters = different entry_id (both stored)

```
# Store an entry and capture the returned key
key = llmcache.store(
    prompt="What is the capital of France?",
    response="Paris",
    metadata={"source": "geography"}
)
print(f"Full Redis key: {key}")
```

```
Full Redis key: llmcache:115049a298532be2f181edb03f766770c0db84c22aff39003fec340deaec7545
```

```
# Check and see both entry_id and key in the response
result = llmcache.check(
    prompt="What is the capital of France?",
    return_fields=["entry_id", "prompt", "response"]
)
print(f"Entry ID: {result[0]['entry_id']}")
print(f"Key: {result[0]['key']}")
```

```
Entry ID: 115049a298532be2f181edb03f766770c0db84c22aff39003fec340deaec7545
Key: llmcache:115049a298532be2f181edb03f766770c0db84c22aff39003fec340deaec7545
```

Fetch and Delete Specific Entries

You can fetch or delete specific cache entries using the underlying index. First, let's see what keys exist in the cache:

```
# List all keys in the cache using the underlying index
from redisvl.query import FilterQuery
from redisvl.query.filter import FilterExpression

query = FilterQuery(
    filter_expression=FilterExpression("*"),
    return_fields=["entry_id", "prompt", "response"]
)
all_entries = llmcache._index.query(query)
print(f"Found {len(all_entries)} entries in cache:")
for entry in all_entries:
    print(f" - entry_id: {entry['entry_id'][:20]}... prompt: {entry['prompt'][:30]}...")
```

```
Found 1 entries in cache:
 - entry_id: 115049a298532be2f181... prompt: What is the capital of France?...
```

```
# Fetch a specific entry by its entry_id
entry_id = result[0]['entry_id']
record = llmcache._index.fetch(entry_id)
print(f"Fetch record: {record}")
```

```
Fetch record: {'metadata': '{"source": "geography"}', 'entry_id':
→ '115049a298532be2f181edb03f766770c0db84c22aff39003fec340deaec7545', 'inserted_at':
→ '1776288885.186736', 'updated_at': '1776288885.1867368', 'prompt': 'What is the_
→ capital of France?', 'response': 'Paris', 'prompt_vector': b'\xa15\x06@\x1c\x10\xba\xbf
→ $\xa28\xbf[\xa8aU\xc0\x02E\x89?q\x825\xbd\xca\xc1\xfb\x186\xfb\xbe\x5\n\x12\xc0\
→ x9es\x83\xc0\xe2'\''?\xb7T\xfc\xbf\x9b\xef\xa1\xbf\x0bo\xae\xbf"\xebG>-\x16\x81\xbf\
→ x8e&\x13\xbf\x21_\xbf\x8b\xfb9>?\x96\xdf(\xbf:7\x99\xbf\xfb7\xbd\x8f\xbfLoQ\xbf\x89\
→ xb2M?c\xea\x16\xc0s\x1e\xdc\xbe\xfb2fx\xbf\x8b\xa5B>\xfaM\xef\xfb9\xdd$?(\xa1=\xbf\x1a\
→ xd0\x99\xbf[dd?\x04\x9c\xc9>\xff\xfb9\x0f\xbe0\x90X\xber[H\xc0=. \xb1=\xb00\x90\xbd\xba\
→ xaa\x81\xbe\xca<A;\xdb\x8ev@.\''0@$x08\xdf?\xe9\x94\x1c\xc0\xce\x89\xfb9>\xfdf\x04\x24\
→ xbf\x26)U\xbf#h\xea?\xef\xba\x1b?\xb9e\x04\xbe\x0fb\x0e@\xdc\xbdL?\xb8i\xb5?\xd8\xfb0u\
→ xbf\xfb3g\x9b>\n\x2\x83\xbf\xe6\xaa\x83\xc0\xc1\xfb7\''?!xe1\xd8\xbe\xc3\xd3'\xc0)\x9c\
→ x90?xx\x23\xbe\xa5\x9dz?\xf8\x10\x9f75\x99\xc5?\x03\xd5\x9f?\xf2\xc0B?\x08\x9f\xb1>oW|?
→ \xa0\xa4e@\xe9\xed\x90?\xfcf\xe9>\xe8\xe9\xe8\xbf\xc5\x04T\xbf\x9du\xa6\xbe\x14\xa2\
```

(continues on next page)

(continued from previous page)

```

→x98\xbf#\xb7+?\xe9\x8a` \xc0\x18uU?\xbd\x1a<@2~\xbf\xbf&1\xa4?8M\x19@\xb0\x883?\x94\x1f\
→x84\xc0\xea=\xd8=^\x1b\xdc\xbe\x1e\xbd\x14\xc0\x10x\x87?\xde\x5\x7f>;[\xee=N\x1f"? \
→xda\x8ci\xbf\xed\xdb\x07\xbe\x12 Z\xc0i\xfb\x03\xbd\x9f\x82\xea>\x9b=\x0f\xc0\xe6D\xbf\
→xbe\x7f\x91\xb9\xbfU)(\xc0\xdbP\x92\xbd\x6\x86\xbfD\x0>\x1a9\x01@\x9c?\x1a\xbf\
→x9a\x9c9\n\x0gd\'?\xcdqr?3%<\xbf\x8f\xe8\x2\xbe\xbe\xae\x9d\xbf\x4\x0e+\xbf-e\x02@4+\
→x9a\xbf\xa5\x88\xe7?W\xe9K\xbe\x7;Y?\xd9\x8a\x94?\r\xaf\x2?*\xd1\xa6?\xb7\x82\x00\
→xc0\xb2\x8d\xbf>X\xd2\x8f?\x1c\xe7\xcf?A\x89J?\xcem\xe4\xbfP2\x15\xbd\x3I9\xbe)\x14H?\
→xbe\x87\x87\xbe\x6\xcb\xa2?Un\xc4\xbf\x2\xaf^>\x1b%\xd6\xbe\x1e\tW?\xcf\x82\x0f@ \
→xa5K\xc8\xbeD\xe1\x90=\xfd\xa4\x12\xbf\x05\xcb>\xbe\x3\xcf\x08\xbf\xe4}\xc4\xbev\x02\
→x92\xbe\x07\x7\x19\xbc\x5\x9b\x9d?\xdev\x03@\x9a\xbff\xbe\xdd\x94\x8f>Y\xad\x8d\xbe\
→x9f\xcf$\xbf0\x0cz\xbe\x1d\xfd\x96\xbf\x7f?n\xbfh_\x12\xc0\x02v\xdd\xbf^E\xa4>\xda{\
→xe3?l\xa5\xab\xbe\xad.\x12@T\xed>\xbf]\xe9Y\xbf\x5\xad\xca\xbd\x8\x01\x1e@\x0f\xbf=\
→xbf\x8b\xe2\xa0>Pf\x98\xbd>\x00\xbe\x6\x5g\xbf\xec~\xa9@\xb6\x10L\xc0Q\x1fA?\xeb!\
→xa6\xbd^u?\xb6\xabi\xbfm\x02\x11\xc0\xa3\x0e\x7?\x9c\x6%\xc0\x9d\x85\x94\xbf\x04\
→xa98>\x9a\xe8_\xbe\x86\xbd\xbf\xa9p\x8e\xbf6G0\xbb\x1\x9c\xa3?\xe36$\xbf\x040\x05\
→xc0r\x72\xa8>\x9a\xe3\x14?\x07M\x8c<S>\xe8\xbfR@\x13<\xfaZ\xe0?l\x00\xde\xbf\xfc\xa5\
→x03?\xd0\x5e.\xc03m\xdc\xbf\x9f\xa7\x90?eB\x97\xbeqE\x04@\xfc\x07<?\xe9\xbe3\xc0\x1eG\
→xa8?\xd7\x9f\x8d\xbfvmi?J\xe4g\xbfj:\xa3?\xcb\x16\x1d\xbfP\x04\xee?/e\x9e>\x8cU\x98?\
→xa8s\xc6=\xf5/\xc1?\x0f1\x80\xcd\xfa0\xbf\x81#\xf0?z4\x03\xbf\x95\x881\xbf\xbf\xbe\'@\
→$\x9c\x5?\x82\n)\xbfxd\x17\xc0\x02EQ?\x83\x7\x83\xbf\x8f\xcc\x84\xbe\xa9\xa0Q\xbf:;\
→xf9?\xdc>\x17\xbf\x02$\xe1?\xf4\x8e\x1b\xbf\x1c\x19:\xc0\x81\x4k?#\x83\x85?\x9a\x8c\
→xdd?\xa3\x10\xa9\xbe\x69\xff>s\x81T=T\xf7\x85\xbfF4\x6\xbe\x10\xf5\x1c@vf\x93\xbf\
→xb4\xa8\x8d?\xd5>\x1f\xc0\xe3\xc0k\xbf/WA\xbf89\xcd?\xe1[\x94?\x81$\xa4\xbe&\' \xd6\xbf,
→: [@\xad/\xd2\xbf\xba\x17\x86?\xc4\xe2\x01\xc0\xa0\x9f\x9f\x0\xbe]\x03\xec\xbdv\xe3\x0f@ \
→xdd\xfb\x13@\xdc\xe2\xa8\xc0\x88c?]\xb4\xb4K\xbf\xd69\x1a?.\xc1\xe1\xbf\xaaNd? \tJE@ \
→x9bB\t\xc0\xa7,\x95;\xa5\xde\x0b\xbe\xa9)>;\xd7\x4\xbf\x11\xe7\x0e?8\x9c\x9d\xbd}\
→xfd\x0c\x0G\xa1\x7f?\xc7\x14\x19?\x8bi\xc1?\x88\xfaS?\x8dg\x19@\x16\x9f\x9e?\x7fa0\
→xbf\x96\x0c\xe8>\xec\x03\x9?Z>\xf0>\xcf\xaa\xda\xbf1WI\xbed\x9c\x9c\xbf$ \x13?:\x02\
→x9b?\t\xe9G?\xf8\x10\x07@1\xe1\xab>\xca\xa6\xdc\xbf\x1b0\xac\xbf\xb9\xb7\x2?1.\xbd>\
→x92\xa0\xf4\xbf!W\x03\xbf\x881\xdd>o\xa1-@\xc7kb?b\x0e\xfd>\x12\xb5\xb7\xbf\x02,\xfa?,\
→x84t?\' \xa3\x0e\xbfT\xb7\xd6?\x89\xb9w\xbe`#\xd4=\xfcC=@\x8b\xa68@\xe8\x88f\xbfV\x0c\
→xd7\xbe\x12\xe8\x0b?=A\x82\xbf\x83\x15\xa9?ogq\xbf\xa1<\xca\xbe\x8e\n\x13\xc0\xbe\x95\
→xce\xbd\x075!\xc09\xa8\xdd?\xec\xce\x9b?\xfd\x8f3\xbf\x6\x8\x7f2?r0\x0e@\x18\x0e\x0\
→x00u\' \xbf\x8f\xca\x90\xbe5C\x11\xbf\x1b\xe1\xcb?Z\x94\x96\xbf!\r\x0H\x83\t\xc0\x90\
→xf6\x1c\xc0\x84M\x1f\xbf\x0e\x0b\x0c\xbf\x0\x1"\xbe\n8?\xc0\x8ad\x1a?\x9a\x4\xbb>\
→xee\x99\xad\xbf{z\xbc\xbf\xcc\xfb\x03\xbe\xffI~\xbd\xd6d\x82>q\x0c\x5?\x17\x0b\x08\
→xc0\x07|C@\xb80{>\xd9+\x1c>\x11d\x12\xc0%,\xc0\x1d \x9b\xbf\x82\x84S?XE\x90?\xba\x52?
→;l\x06@Z\xfd\x80?\x1a\x97\xac?Fn\xcc\xbf\x4I\x1c?\x9b\xa7\xdf?\xfe0s\xbf\xacN\xa6=B\
→xfe#==\xfe"? \xa8\\\x1d\xbf\xbd(\xf0?\xa1~<\xbe\xfd\xda\x8f>\x83\x19\xab\xbe\xcc:\x0e\
→xhc\xba\xbe\xe3e\xef?\xdag\x8d?\x82\x1f\xfe\xbf\xcb\x4\x06\xbd6CD@\xfeqq\xbf\xce\x0\
→x96?\x89 M?\x9b)\x87\xbe\x934(\xc0W\xba\xa0\xbe\xeb\x84s>\x1fi\xfc?\xfc\x0fs\xbf\x9\
→xfd\x98?Y\xb7\x9e>\x18\xfe\x00\xbe^)\n?\xa8\xb4\xff\xbe\xec\xde)=Y\x9b\x89@M\x9a\x5\
→xbe\xef\xa2G?\xf3\xa3\x8b\xbf[\xf9\x8b\xbfw\xb5\x86?\xcb\x16\x04\x04\xd6\xa3\xbf\x09\
→x9a?}\xd5!@\xdb\x5\x87?\xcfc(\x86?\xaa\x80\x1e\xc0\x14&\xa6\xbf\x18\x11?\x88IH\xbf\
→x82CH?\xb6G\x17@9\xa9&\xbe#Y\x8c>\xa5\x9\x0e\xc0\x15\x93*?\x97\xe5\x02?\xf0\xcaW\xbf\
→xfeh&\xc0C\x7f\r@\x0c\x0Z\xbf\x91DM@\nzm?R\x4\x6<\x7fv\x4\xbfA\x96\x91\xbeg\x3\
→x8e>\xd8\x3!?\xf1\x838\xc0\x9^ \xbf?|@{>\xf2@\x99?\x96hj\xbf\x86\x2\xbe?3\x10m\
→xc0\x0f\xe9\xcb?\xc6\x2\x9c?\xb2\x4\x9d?-\xa6A\xbf*\xe5\xed?D\x01=\xb2\x5\x9d\xbf\
→t\x00\n\xc0\xca\xecb>f\x6r\xc0\xa4\' \xe7\xbf\xea\x00\xa0\xbfD\xa03?9\x6\x04@\x1b\x07\
→xbd?\x18^ \xa8\xbf[\xcfc\x8=L\xfd\xa1\xbes;c\xbe\x1e7*\xbf\x8~\xb7\xbd\x96\xa2\xaa<\
→x8e\xc0k?&\xf0\x1a\xbf\x1f\x8cM\xc0\xb1)\xe7?\xe4\xfeu\xbf\x1d\x2"@ \xf6\xa5%y\xdf\
→xe9?\x87S\xae?\x8e\xef.\xc0\x9\x91\xb5\xbf\x00\xdb\x12\xbf\x05}#\xbd\x1d\xa8\x16\xbf%\

```

(continues on next page)

(continued from previous page)

```

→xcd\x1a\xfb\x9dZ\xfb\xde\xbfC\xfb\x1\xee\x18?\xa7\x1b\x9b\xbf(j\x05@\xbd\x06r? \xd7\
→x1d>\xbaWL\xfb\x9f\x8\xfc>\x99\xa2)\xc0\xfb\xed\x99\xfb\x1d\xbd\x99\xfb\xea\x80h\xbe\
→x82H\x1d\x0>G\xbc\xfb\x9f\x8\x0e\xfb\xa2\x1aV?\xba\x0f\x14@\xb0\xed\xad?8\x11>\xbei\
→x18\x05\x00\x19\x0c\xfd\xfb8\xfb37@#\x14\x06@F\r\xfb4?\xa8\x9er\xfb?p\x8d\xbe}\xdb\x99\
→\xbfb\x88\xfb6\x19?\xab`w>]\xf6\x00\xfb\x8c\x0d+\xbfb=V\xda?\xab3\x9f?Kq\xec\xbd\x893&\xbfb\
→\xf3A\x01\x0R\xa8\x0c\x0e7\xe5\n@\xd5\x02\x00@\xc9V0@o\xfe\x82\x00\x1a\x8a\x1f\xfb\
→xcd\x81!>\xbb\x17` \xbfb\x16\xe9\xac?\xd9^\x95>): \x91\xbe\x0f\x08h\xfb\x8eR\x09\xfb\xfb\
→'E\xbe\x01\x85z?g\x85\x1b@\xe80G@U\x1b\x04\xbe\xfb9\x87\x8a?\x1a\x83\x01\xfb5\x87\x83>\
→x9c\x86h\xfb` \xd06\xfb\x89\x10\xa7?\xd0\x19\xda\xfb\x10\x81x?6Kh?\x0bT\x03@\xfesd>\xce\
→x1d\xba?\t\xe6\x8b\xbe\x12\x8a\x96\xbe\x16\x84\x8b=OS!\xc0a=\x02@\xba\x06\x95\xbe\xcb\
→xdd\x0b=h\xfb5\x85?7\x8c\x80\xbe&\xd5\x98\xfb\x10\t\xfb1>I\xba\x89\xfb!\xcb7>q0\n@\xf4\
→x8e\x10\xfb86\x84?\xec\n\x06@?\x9b\x8b\xbe\x07\x99\x05?\xb5Zp?,<\xbfb\x01\x98\xfbU\
→xc5\x9d\xbe\x9dS\x19\xfbfhL\x01\xfbfre\xed\xbd:\x0fv\xbe\x07Vf\xbe\x82H\xa7\xfb\x8e\xfb7\
→x04\xfb\x94\xfb1P?\x04K\xe2\xfbf|\xa4\x1c&?5@\xc8\x92\x12\xfb\x09\xe7i>\xfbx85-?\x1cJ]\
→\xbfb\xfb50\x02=\x02\xcdK@\xcd\x9c\x11\x00\x13\x02\x99\xfb\x1b\x09\x0d\x01\x0c]\xbfb\
→\xf6\xcd\x93\xfb\x02\n@R\x02`'\? \xed\xa2\x88?0q\xeb\xfb\x00\x05\x01\xbc/\xc7@?'\xed\
→\xde\xbe\xe3\x95\xdd\xbd\x06\x8b\x8d8?K\x10\xaa>nL\xfd\xbe\x8e\x04\xec?\xea\x07d?M\xae\
→\xeb>\xe9\x9e\xa0\xbe\x87y\x88?\xfdf1\x16?\x1e\x07a?b|+?\xada\xff\xbe\x865V?\x01\x85\
→\xe9\xfb\xfb62(\xc0\x1e/\xcb?\xd7\x8f\x8b=\xcd\x88]\xc0&\xfc\xde?\xb7 \x1f\x00\xe1\x0d\
→\x0b\xfb\x85\x0bI\xfb\x84\x09\x01\xbe\x1b\xea\x84?\xf8\x00\xfbeg\x83=\xa1t\t\x00\xfb\
→\xda1@(\x14\xa5?\xfc\x06\xfb\x00\xbe\n\x1a*@\x04+\x99?\xd8\xa9\xbe?\xc7$\x84?Q]\xcd?E\xfb\
→\x1b\x00\xfdma\xbe\x1d\x83\x12\x00\x8f\xe9\x04=\xcc\xe4\xfa\xbe\x14\xec\xbe\x09\x05\
→\xbe=\xda\x05\x00\x85z\x03?|\x15? \xc3\\\xcac?\xc7QW\xfbIu\xfb4\xbe%***\xb4Q,\xc0y-\x12@\
→\xc1\x0d\x0f@:j\x01@k\x87\xba?b/\xe1\xbe\xbaa<\xb59\x98\xfbfa_\x07@\xd9a\x0c>\x01\x12\
→\xa1?R\x16\x94\xfb\x02\xdb\x03@\xef\x00m\xfbf1:\xb3=~\x95\x19\xfb\x9c\x1b]\xbd^o\x82\xfb\
→\xa1i \xc0\xdd\x8dD\x00\xcc\x04\xe3\xfb\x90\xbb\xba?\xb5\xfb8\xa5\xfbf%\xab\t\x07\x05\
→\x8b?\xa5\x16\xbe?\x0b` \x01\xbe/\xc2e?s\x0c\x88>z\xfb2\x0e>\x97\x93#\xc0h\x9b\r@\x0bJ\
→\x95\xfbf4\x8b\x93>vSZ@\x0c7\xae\xfbf'\x82\x1a\x00\x0d\x1b!\>\x8ag\xa0?G\xcb\xa5>n@\xd5\
→\xbfb\x84\x0f\@2\x88H?\x9bD\x8d?\xc5n\x80\xfb\x07\x08\x9d\xfbfk\x10\xa1>\xa9< \xbe\xba\
→\xc4\x05=\x94\x09\x1f\x00\xdd\x01\x0d>\xa0\x9c\x06?)\x8d}\xbfb\x83\x91\xbeEb\xed<%\xa1aE?
→\xaf\x04\x05\xbe\xbc\xa3 ?\xd3#\x9e?xi\x14?\xe0G#@\xc7\xef \xbfb\xde:\xbfb\x9a\xcf+\xbe\
→\x8b\xa7\xaa?>0\x8e\xfb\xfb1\xdd\x13\x09\xea\xe4\xfbfB\x1d#@*(\t\x00+\x7f\x14@|@\x9e\
→\xbfb\x89\xe4\xfb\x96<L?\x1f\x0e\x15\x05\xfbf\xfbf&P\xae>\x95\xa3\xcb?\xa0z\xa4?6p\xfb6\
→\xbfb!\xd4\x02@\xf2\xbe-\xbd(\xc8\x12\xfb\x8aR"=pGQ>\xc2\x16\xfb\x09\x0d.\xc0\x95\x84\
→\x83?\x94\x13\xcb?\xb4\x14\xee\xfbf*\x8c\x1f@ZK\x1d@\xc2\x08J\xfbfSsz\xfb\x00"\xee\xbe\
→\xd72\x8b6\xbe\x01\x03\x11\xfbf;\xe6?>F\xbcK\xfbfP]\xa7\x00\xbaW\xbe>\xd18%>\xdbo\x1b\x00k\
→\xb8a\xbe\x15\xdd\xaa\xfbfY\xaeW@\x03\x93\x11\x00\xfb3<M?^\x9a\xfb2?U\rU\xfb\x8e\xde\x1f>\
→\x07\xa8\x9c?\x04\x0f\x81\xbeI\x1b\x0d7\xbe\xae\xbe\x1a\xfbf\x82?\xc0\xdd\x83p>\xc3\xa1\
→\xc7=\x12\xa9K>\x1b\x18\xdc=\x92\x97\xe2\xfb\x1ety\xbey\xa2X?k\x03p?\x9b]\xb6?\xce\x8\
→\x1b?uG\x19@'}

```

```
# Delete specific entries by ID or key
```

```
# By entry IDs (without prefix):
```

```
llmcache.drop(ids=[entry_id])
```

```
# Or by full Redis keys (with prefix):
```

```
# llmcache.drop(keys=[key])
```

```
# Verify it's deleted
```

```
result = llmcache.check(prompt="What is the capital of France?")
```

```
print(f"After deletion: {result}")
```

```
After deletion: []
```

Customize the Distance Threshold

For most use cases, the right semantic similarity threshold is not a fixed quantity. Depending on the choice of embedding model, the properties of the input query, and even business use case – the threshold might need to change.

The distance threshold uses Redis COSINE distance units [0-2], where 0 means identical and 2 means completely different.

Fortunately, you can seamlessly adjust the threshold at any point like below:

```
# Widen the semantic distance threshold (allow less similar matches)
llmcache.set_threshold(0.5)

# Re-store an entry for the threshold demo
llmcache.store(prompt="What is the capital of France?", response="Paris")
```

```
'llmcache:115049a298532be2f181edb03f766770c0db84c22aff39003fec340deaec7545'
```

```
# Really try to trick it by asking around the point
# But is able to slip just under our new threshold
question = "What is the capital of the country where Nice is located?"
llmcache.check(prompt=question)[0]['response']
```

```
'Paris'
```

```
# Invalidate the cache completely by clearing it out
llmcache.clear()

# Should be empty now
llmcache.check(prompt=question)
```

```
[]
```

Utilize TTL

Redis uses TTL policies (optional) to expire individual keys at points in time in the future. This allows you to focus on your data flow and business logic without bothering with complex cleanup tasks.

A TTL policy set on the `SemanticCache` allows you to temporarily hold onto cache entries. Below, the TTL policy is set to 5 seconds.

```
llmcache.set_ttl(5) # 5 seconds
```

```
llmcache.store("This is a TTL test", "This is a TTL test response")

time.sleep(6)
```

```
# confirm that the cache has cleared by now on it's own
result = llmcache.check("This is a TTL test")

print(result)
```

```
[]
```

```
# Reset the TTL to null (long lived data)
llmcache.set_ttl()
```

TTL Behavior Details

Understanding how TTL works with SemanticCache is important for production use:

Scenario	Behavior
ttl=None (default)	Entries persist forever. <code>check()</code> does not affect TTL.
ttl=3600 at init	Entries get TTL on <code>store()</code> . TTL is refreshed on every <code>check()</code> hit.
Set TTL later with <code>set_ttl(3600)</code>	Existing entries keep no TTL , but <code>check()</code> will now add TTL to matched entries.
Remove TTL with <code>set_ttl(None)</code>	Existing entries keep their TTL, but <code>check()</code> no longer refreshes it.

Important: The `check()` method automatically refreshes TTL on all matched entries (sliding window pattern). This keeps frequently accessed entries alive but can unexpectedly add TTL to entries that were originally stored without one.

TTL Refresh on Check

When `check()` finds matching entries, it refreshes the TTL on **all** matched results, not just the one you use:

```
# Example: TTL refresh behavior
llmcache.set_ttl(300) # 5 minutes

# Store an entry
llmcache.store("What is Python?", "A programming language")

# Every time check() finds this entry, TTL resets to 300 seconds
result = llmcache.check("What is Python?") # TTL refreshed

# Reset for next examples
llmcache.set_ttl()
llmcache.clear()
```

Simple Performance Testing

Next, measure the speedup obtained by using SemanticCache. The `time` module measures the time taken to generate responses with and without SemanticCache.

```
def answer_question(question: str) -> str:
    """Helper function to answer a simple question using OpenAI with a wrapper
    check for the answer in the semantic cache first.

    Args:
        question (str): User input question.

    Returns:
        str: Response.
    """
    results = llmcache.check(prompt=question)
    if results:
        return results[0]["response"]
    else:
        answer = ask_openai(question)
        return answer
```

```
start = time.time()
# asking a question -- openai response time
question = "What was the name of the first US President?"
answer = answer_question(question)
end = time.time()

print(f"Without caching, a call to openAI to answer this simple question took {end-start}
↪ seconds.")

# add the entry to our LLM cache
llmcache.store(prompt=question, response="George Washington")
```

```
Without caching, a call to openAI to answer this simple question took 1.346540927886963_
↪ seconds.
```

```
'llmcache:67e0f6e28fe2a61c0022fd42bf734bb8ffe49d3e375fd69d692574295a20fc1a'
```

```
# Calculate the avg latency for caching over LLM usage
times = []

for _ in range(10):
    cached_start = time.time()
    cached_answer = answer_question(question)
    cached_end = time.time()
    times.append(cached_end-cached_start)

avg_time_with_cache = np.mean(times)
print(f"Avg time taken with LLM cache enabled: {avg_time_with_cache}")
print(f"Percentage of time saved: {round(((end - start) - avg_time_with_cache) / (end -
↪ start) * 100, 2)}%")
```

```
Avg time taken with LLM cache enabled: 0.04209451675415039
Percentage of time saved: 96.87%
```

```
# check the stats of the index
!rvl stats -i llmcache
```

Statistics:

Stat Key	Value
num_docs	1
num_terms	24
max_doc_id	9
num_records	83
percent_indexed	1
hash_indexing_failures	0
number_of_uses	23
bytes_per_record_avg	36.7469863
doc_table_size_mb	0.01546192
inverted_sz_mb	0.00290870
key_table_size_mb	2.76565551
offset_bits_per_record_avg	8
offset_vectors_sz_mb	5.72204589
offsets_per_term_avg	0.72289156
records_per_doc_avg	83
sortable_values_size_mb	0
total_indexing_time	4.01000022
total_inverted_index_blocks	26
vector_index_sz_mb	3.01629638

```
# Clear the cache AND delete the underlying index
llmcache.delete()
```

Cache Access Controls, Tags & Filters

When running complex workflows with similar applications, or handling multiple users it's important to keep data segregated. Building on top of RedisVL's support for complex and hybrid queries we can tag and filter cache entries using custom-defined `filterable_fields`.

Let's store multiple users' data in our cache with similar prompts and ensure we return only the correct user information:

```
private_cache = SemanticCache(
    name="private_cache",
    filterable_fields=[{"name": "user_id", "type": "tag"}]
)

private_cache.store(
    prompt="What is the phone number linked to my account?",
    response="The number on file is 123-555-0000",
    filters={"user_id": "abc"},
```

(continues on next page)

(continued from previous page)

```
)

private_cache.store(
    prompt="What's the phone number linked in my account?",
    response="The number on file is 123-555-1111",
    filters={"user_id": "def"},
)
```

You try to use a model that was created with version 4.1.0, however, your version is 3.4.1. This might cause unexpected behavior or errors. In that case, try to update to the latest version.

```
'private_cache:2831a0659fb888e203cd9fedb9f65681bfa55e4977c092ed1bf87d42d2655081'
```

```
from redisvl.query.filter import Tag

# define user id filter
user_id_filter = Tag("user_id") == "abc"

response = private_cache.check(
    prompt="What is the phone number linked to my account?",
    filter_expression=user_id_filter,
    num_results=2
)

print(f"found {len(response)} entry \n{response[0]['response']}")
```

```
found 1 entry
The number on file is 123-555-0000
```

```
# Cleanup
private_cache.delete()
```

Multiple `filterable_fields` can be defined on a cache, and complex filter expressions can be constructed to filter on these fields, as well as the default fields already present.

```
complex_cache = SemanticCache(
    name='account_data',
    filterable_fields=[
        {"name": "user_id", "type": "tag"},
        {"name": "account_type", "type": "tag"},
        {"name": "account_balance", "type": "numeric"},
        {"name": "transaction_amount", "type": "numeric"}
    ]
)

complex_cache.store(
    prompt="what is my most recent checking account transaction under $100?",
    response="Your most recent transaction was for $75",
    filters={"user_id": "abc", "account_type": "checking", "transaction_amount": 75},
)

complex_cache.store(
```

(continues on next page)

(continued from previous page)

```

prompt="what is my most recent savings account transaction?",
response="Your most recent deposit was for $300",
filters={"user_id": "abc", "account_type": "savings", "transaction_amount": 300},
)
complex_cache.store(
    prompt="what is my most recent checking account transaction over $200?",
    response="Your most recent transaction was for $350",
    filters={"user_id": "abc", "account_type": "checking", "transaction_amount": 350},
)
complex_cache.store(
    prompt="what is my checking account balance?",
    response="Your current checking account is $1850",
    filters={"user_id": "abc", "account_type": "checking"},
)

```

You try to use a model that was created with version 4.1.0, however, your version is 3.4.
 ↳ 1. This might cause unexpected behavior or errors. In that case, try to update to the
 ↳ latest version.

```
'account_data:944f89729b09ca46b99923d223db45e0bccf584cfd53fcdf87d2a58f072582d3'
```

```

from redisvl.query.filter import Num

value_filter = Num("transaction_amount") > 100
account_filter = Tag("account_type") == "checking"
complex_filter = value_filter & account_filter

# check for checking account transactions over $100
complex_cache.set_threshold(0.3)
response = complex_cache.check(
    prompt="what is my most recent checking account transaction?",
    filter_expression=complex_filter,
    num_results=5
)
print(f'found {len(response)} entry')
print(response[0]["response"])

```

```

found 1 entry
Your most recent transaction was for $350

```

Next Steps

Now that you understand semantic caching, explore these related guides:

- *Cache Embeddings* - Cache embedding vectors for faster repeated computations
- *Manage LLM Message History* - Store and retrieve conversation history
- *Query and Filter Data* - Learn more about filter expressions for cache access control

Cleanup

```
complex_cache.delete()
```

Use LangCache as the LLM Cache Backend

This guide shows how to use RedisVL's `LangCacheSemanticCache`, a thin wrapper around the [LangCache](#) managed semantic cache service. You get the same high-level `check` / `store` workflow as `SemanticCache`, backed by LangCache's HTTP API instead of a Redis index you manage yourself.

For more on semantic caching, see [Extensions](#), and to use RedisVL's semantic caching class see our [llm cache notebook](#). API entries for both classes live in the [LLM cache API](#).

Prerequisites

Before you begin, ensure you have:

- Installed RedisVL with the LangCache extra: `pip install redisvl[langcache]`
- Python 3.10+ (same as RedisVL)
- A LangCache service with a **cache ID** and **API key**. You can set up a LangCache service in Redis Cloud [here](#)
- Optionally: **attributes** configured on your LangCache cache if you plan to pass `metadata` / `attributes` from RedisVL

What You'll Learn

By the end of this guide, you will be able to:

- Choose between `SemanticCache` and `LangCacheSemanticCache` for your deployment
- Initialize `LangCacheSemanticCache` with credentials and TTL defaults
- Implement read-through caching (`check` → LLM → `store`)
- Use LangCache attributes for scoping and deletion
- Override TTL per store, use async APIs, and run delete operations
- Understand current limitations compared to `SemanticCache`

Choose SemanticCache or LangCacheSemanticCache

	SemanticCache	LangCacheSemanticCache
Where data lives	Your Redis deployment; RedisVL creates and queries a search index	LangCache managed service (hosted API)
Best when	You control Redis, need full RedisVL query/filter features, or co-locate cache with app data	You want a managed semantic cache without operating Redis or the index
Vector search by raw embedding	Supported (vector= on check)	Not supported — search is prompt-based via the LangCache API
Filter expressions	FilterExpression on check	Not supported — use LangCache attributes (pre-configured on the cache)
Partial entry updates	Supported where the backend allows	update / aupdate raise — delete and re-store instead

Note: SemanticCache is covered in depth in the *llmcache notebook* guide.

Install the LangCache extra

The `redisvl[langcache]` extra installs compatible `langcache` dependencies:

```
pip install redisvl[langcache]
```

```
# NBVAL_SKIP
%pip install redisvl[langcache]
```

Initialize LangCacheSemanticCache

Create `LangCacheSemanticCache` with your LangCache credentials. The default `server_url` points at the managed LangCache API; override it if your provider gives a different endpoint.

The following example reads credentials from environment variables (recommended for applications). Replace placeholder values when experimenting locally.

```
# NBVAL_SKIP
import os

from redisvl.extensions.cache.llm import LangCacheSemanticCache

CACHE_ID = os.environ.get("LANGCACHE_CACHE_ID", "YOUR_CACHE_ID")
API_KEY = os.environ.get("LANGCACHE_API_KEY", "YOUR_API_KEY")

cache = LangCacheSemanticCache(
    name="my_app_cache",
    server_url="https://aws-us-east-1.langcache.redis.io",
    cache_id=CACHE_ID,
    api_key=API_KEY,
    ttl=3600, # default TTL for entries, in seconds (optional)
)
```

Parameter	Purpose
cache_id, api_key	Required. Identify your LangCache cache and authenticate.
server_url	LangCache API base URL (default matches typical managed deployments).
ttl	Default time-to-live for stored entries, in seconds; can be overridden per store call.
use_exact_search / use_semantic_search	Enable exact and/or semantic matching (at least one must be True).
distance_threshold (on check)	Works with distance_scale: "normalized" (0–1 distance) or "redis" (cosine-style 0–2).

Attributes and metadata

LangCache **attributes** are key/value metadata attached to entries. They can be used when **searching** (check / acheck via the attributes argument) and when **deleting** (delete_by_attributes / adelete_by_attributes).

You must define the same attribute names (and types) in the LangCache console or API for your cache before RedisVL can use them. If you pass metadata to store or attributes to check but the cache has no attributes configured, the LangCache API returns an error; RedisVL surfaces a clear `RuntimeError` explaining that attributes need to be configured or removed from the call.

String values are encoded for the API and decoded when reading hits so special characters remain usable.

Read-through caching pattern

Typical flow: try check, call the LLM on a miss, then store the result.

```
# NBVAL_SKIP
def call_your_llm(prompt: str) -> str:
    """Replace with your LLM client (OpenAI, Anthropic, etc.)."""
    return f"Answer for: {prompt}"

def answer(user_prompt: str) -> str:
    hits = cache.check(prompt=user_prompt, num_results=1)
    if hits:
        return hits[0]["response"]

    response = call_your_llm(user_prompt)
    cache.store(prompt=user_prompt, response=response)
    return response
```

Optional scoping with attributes (only if those attributes are configured on LangCache):

```
# NBVAL_SKIP
user_prompt = "Example prompt"

hits = cache.check(
    prompt=user_prompt,
    attributes={"tenant_id": "acme", "model": "gpt-4o"},
```

(continues on next page)

(continued from previous page)

```
num_results=1,
)
```

TTL

- **Constructor** `ttl=` sets the default lifetime for new entries (seconds).
- **Per call**, pass `ttl=` to `store` / `astore` to override the default for that entry.

```
# NBVAL_SKIP
prompt = "What is Redis?"
response = "Redis is an in memory data store."

cache.store(prompt=prompt, response=response, ttl=300) # this entry expires in 5 minutes
```

Async usage

Use the `a-`prefixed methods with `asyncio` — for example `acheck`, `astore`, `adelete`, `adelete_by_id`, `adelete_by_attributes`, `aclear`.

```
# NBVAL_SKIP
async def call_your_llm_async(prompt: str) -> str:
    return f"Async answer for: {prompt}"

async def answer_async(user_prompt: str) -> str:
    hits = await cache.acheck(prompt=user_prompt, num_results=1)
    if hits:
        return hits[0]["response"]

    response = await call_your_llm_async(user_prompt)
    await cache.astore(prompt=user_prompt, response=response)
    return response
```

Delete operations

Method	What it does
<code>delete()</code> / <code>adelete()</code>	Flush the entire cache (all entries). Aliases: <code>clear()</code> / <code>aclear()</code> .
<code>delete_by_id(entry_id)</code> <code>adelete_by_id</code>	/ Remove one entry by LangCache entry ID (returned from <code>store</code>).
<code>delete_by_attributes</code> <code>adelete_by_attributes</code>	/ Remove entries matching the given attribute map (non-empty dict required).

Current limitations

The wrapper follows the LangCache API. The following RedisVL features either do not apply or are explicitly unsupported:

- **No direct vector search** — Passing `vector=` to `check` / `acheck` logs a warning and does not search by embedding.
- **No `filter_expression`** — RedisVL filter expressions are not translated; use LangCache attributes only.
- **No `update()` / `auupdate()`** — The LangCache API does not update individual entries; these methods raise `NotImplementedError`. Delete the entry (or store a new pair) instead.
- **filters on store** — Not supported by LangCache; a warning is logged if provided.

Tip: See the `LangCacheSemanticCache` section in the *LLM cache API* for parameter and method listings.

Manage LLM Message History

Large Language Models are inherently stateless with no knowledge of previous interactions. This becomes a challenge when engaging in long conversations that rely on context. The solution is to store and retrieve conversation history with each LLM call.

This guide demonstrates how to use Redis to structure, store, and retrieve conversational message history.

Prerequisites

Before you begin, ensure you have:

- Installed RedisVL: `pip install redisvl`
- A running Redis instance ([Redis 8+](#) or [Redis Cloud](#))

What You'll Learn

By the end of this guide, you will be able to:

- Store and retrieve conversation messages with `MessageHistory`
- Manage multiple users and conversations with session tags
- Use `SemanticMessageHistory` for relevance-based context retrieval
- Prune incorrect or unwanted messages from conversation history

```
from redisvl.extensions.message_history import MessageHistory

chat_history = MessageHistory(name='student tutor')
```

To align with common LLM APIs, Redis stores messages with `role` and `content` fields. The supported roles are “system”, “user” and “llm”.

You can store messages one at a time or all at once.

```
chat_history.add_message({"role": "system", "content": "You are a helpful geography tutor,↵
↪giving simple and short answers to questions about European countries."})
chat_history.add_messages([
    {"role": "user", "content": "What is the capital of France?"},
    {"role": "llm", "content": "The capital is Paris."},
    {"role": "user", "content": "And what is the capital of Spain?"},
    {"role": "llm", "content": "The capital is Madrid."},
    {"role": "user", "content": "What is the population of Great Britain?"},
    {"role": "llm", "content": "As of 2023 the population of Great Britain is↵
↪approximately 67 million people."},
])
```

At any point we can retrieve the recent history of the conversation. It will be ordered by entry time.

```
context = chat_history.get_recent()
for message in context:
    print(message)
```

```
{'role': 'llm', 'content': 'The capital is Paris.'}
{'role': 'user', 'content': 'And what is the capital of Spain?'}
{'role': 'llm', 'content': 'The capital is Madrid.'}
{'role': 'user', 'content': 'What is the population of Great Britain?'}
{'role': 'llm', 'content': 'As of 2023 the population of Great Britain is approximately↵
↪67 million people.'}
```

In many LLM flows, conversations progress through a series of prompt and response pairs. MessageHistory provides a store() convenience function to add these efficiently.

```
prompt = "what is the size of England compared to Portugal?"
response = "England is larger in land area than Portal by about 15000 square miles."
chat_history.store(prompt, response)
```

```
context = chat_history.get_recent(top_k=6)
for message in context:
    print(message)
```

```
{'role': 'user', 'content': 'And what is the capital of Spain?'}
{'role': 'llm', 'content': 'The capital is Madrid.'}
{'role': 'user', 'content': 'What is the population of Great Britain?'}
{'role': 'llm', 'content': 'As of 2023 the population of Great Britain is approximately↵
↪67 million people.'}
{'role': 'user', 'content': 'what is the size of England compared to Portugal?'}
{'role': 'llm', 'content': 'England is larger in land area than Portal by about 15000↵
↪square miles.'}
```

Managing multiple users and conversations

For applications that need to handle multiple conversations concurrently, Redis supports tagging messages to keep conversations separated.

```
chat_history.add_message({"role": "system", "content": "You are a helpful algebra tutor,
↳ giving simple answers to math problems."}, session_tag='student two')
chat_history.add_messages([
    {"role": "user", "content": "What is the value of x in the equation 2x + 3 = 7?"},
    {"role": "llm", "content": "The value of x is 2."},
    {"role": "user", "content": "What is the value of y in the equation 3y - 5 = 7?"},
    {"role": "llm", "content": "The value of y is 4."}],
    session_tag='student two'
)

for math_message in chat_history.get_recent(session_tag='student two'):
    print(math_message)
```

```
{'role': 'system', 'content': 'You are a helpful algebra tutor, giving simple answers to
↳ math problems.'}
{'role': 'user', 'content': 'What is the value of x in the equation 2x + 3 = 7?'}
{'role': 'llm', 'content': 'The value of x is 2.'}
{'role': 'user', 'content': 'What is the value of y in the equation 3y - 5 = 7?'}
{'role': 'llm', 'content': 'The value of y is 4.'}
```

Semantic message history

For longer conversations our list of messages keeps growing. Since LLMs are stateless we have to continue to pass this conversation history on each subsequent call to ensure the LLM has the correct context.

A typical flow looks like this:

```
while True:
    prompt = input('enter your next question')
    context = chat_history.get_recent()
    response = LLM_api_call(prompt=prompt, context=context)
    chat_history.store(prompt, response)
```

This works, but as context keeps growing so too does our LLM token count, which increases latency and cost.

Conversation histories can be truncated, but that can lead to losing relevant information that appeared early on.

A better solution is to pass only the relevant conversational context on each subsequent call.

For this, RedisVL has the `SemanticMessageHistory`, which uses vector similarity search to return only semantically relevant sections of the conversation.

```
from redisvl.extensions.message_history import SemanticMessageHistory
semantic_history = SemanticMessageHistory(name='tutor')

semantic_history.add_messages(chat_history.get_recent(top_k=8))
```

```
prompt = "what have I learned about the size of England?"
semantic_history.set_distance_threshold(0.35)
context = semantic_history.get_relevant(prompt)
for message in context:
    print(message)
```

```
{'role': 'user', 'content': 'what is the size of England compared to Portugal?'}
```

You can adjust the degree of semantic similarity needed to be included in your context.

Setting a distance threshold close to 0.0 will require an exact semantic match, while a distance threshold of 2.0 will include everything (Redis COSINE distance range is [0-2]).

```
semantic_history.set_distance_threshold(0.7)

larger_context = semantic_history.get_relevant(prompt)
for message in larger_context:
    print(message)
```

```
{'role': 'user', 'content': 'what is the size of England compared to Portugal?'}
{'role': 'llm', 'content': 'England is larger in land area than Portal by about 15000_
↳square miles.'}
{'role': 'user', 'content': 'What is the population of Great Britain?'}
{'role': 'llm', 'content': 'As of 2023 the population of Great Britain is approximately_
↳67 million people.'}
{'role': 'user', 'content': 'And what is the capital of Spain?'}
```

Conversation control

LLMs can hallucinate on occasion and when this happens it can be useful to prune incorrect information from conversational histories so this incorrect information doesn't continue to be passed as context.

```
semantic_history.store(
    prompt="what is the smallest country in Europe?",
    response="Monaco is the smallest country in Europe at 0.78 square miles." #_
↳Incorrect. Vatican City is the smallest country in Europe
)

# get the key of the incorrect message
context = semantic_history.get_recent(top_k=1, raw=True)
bad_key = context[0]['entry_id']
semantic_history.drop(bad_key)

corrected_context = semantic_history.get_recent()
for message in corrected_context:
    print(message)
```

```
{'role': 'user', 'content': 'What is the population of Great Britain?'}
{'role': 'llm', 'content': 'As of 2023 the population of Great Britain is approximately_
↳67 million people.'}
{'role': 'user', 'content': 'what is the size of England compared to Portugal?'}
```

(continues on next page)

(continued from previous page)

```
{'role': 'llm', 'content': 'England is larger in land area than Portal by about 150000↵  
↵square miles.'}  
{'role': 'user', 'content': 'what is the smallest country in Europe?'}
```

Retrieving message counts

To get the total number of messages stored in a session, use the `.count()` method. You can optionally pass a `session_tag` argument to retrieve the count for a different conversation session.

```
print(f"Total messages in the session: {chat_history.count()}")
```

```
Total messages in the session: 7
```

Next Steps

Now that you understand message history management, explore these related guides:

- [Cache LLM Responses](#) - Reduce API costs with semantic caching
- [Route Queries with SemanticRouter](#) - Classify user queries to routes
- [Create Embeddings with Vectorizers](#) - Use different embedding providers

Cleanup

```
chat_history.clear()  
semantic_history.clear()
```

Route Queries with SemanticRouter

RedisVL provides a `SemanticRouter` interface that uses Redis' built-in search and aggregation to perform KNN-style classification over a set of `Route` references to determine the best match.

This guide covers how to use Redis as a Semantic Router for your applications.

Prerequisites

Before you begin, ensure you have:

- Installed RedisVL: `pip install redisvl`
- A running Redis instance ([Redis 8+](#) or [Redis Cloud](#))

What You'll Learn

By the end of this guide, you will be able to:

- Define routes with references and distance thresholds
- Initialize and configure a `SemanticRouter`
- Route queries to single or multiple matching routes
- Serialize and restore router configurations
- Manage route references dynamically

Define the Routes

Below we define 3 different routes. One for `technology`, one for `sports`, and another for `entertainment`. Now for this example, the goal here is surely topic “classification”. But you can create routes and references for almost anything.

Each route has a set of references that cover the “semantic surface area” of the route. The incoming query from a user needs to be semantically similar to one or more of the references in order to “match” on the route.

Additionally, each route has a `distance_threshold` which determines the maximum distance between the query and the reference for the query to be routed to the route. This value is unique to each route and uses Redis COSINE distance units (0-2], where lower values require stricter matching.

```
from redisvl.extensions.router import Route

# Define routes for the semantic router
technology = Route(
    name="technology",
    references=[
        "what are the latest advancements in AI?",
        "tell me about the newest gadgets",
        "what's trending in tech?"
    ],
    metadata={"category": "tech", "priority": 1},
    distance_threshold=0.71
)

sports = Route(
    name="sports",
    references=[
        "who won the game last night?",
        "tell me about the upcoming sports events",
        "what's the latest in the world of sports?",
        "sports",
        "basketball and football"
    ],
    metadata={"category": "sports", "priority": 2},
    distance_threshold=0.72
)

entertainment = Route(
    name="entertainment",
    references=[
```

(continues on next page)

(continued from previous page)

```
        "what are the top movies right now?",
        "who won the best actor award?",
        "what's new in the entertainment industry?"
    ],
    metadata={"category": "entertainment", "priority": 3},
    distance_threshold=0.7
)
```

Initialize the SemanticRouter

SemanticRouter will automatically create an index within Redis upon initialization for the route references. By default, it uses the HFTextVectorizer to generate embeddings for each route reference.

```
import os
from redisvl.extensions.router import SemanticRouter
from redisvl.utils.vectorize import HFTextVectorizer

os.environ["TOKENIZERS_PARALLELISM"] = "false"

# Initialize the SemanticRouter
router = SemanticRouter(
    name="topic-router",
    vectorizer=HFTextVectorizer(),
    routes=[technology, sports, entertainment],
    redis_url="redis://localhost:6379",
    overwrite=True # Blow away any other routing index with this name
)
```

```
# look at the index specification created for the semantic router
!rvm index info -i topic-router
```

Index Information:					
Index Name	Storage Type	Prefixes	Index Options	Indexing	
topic-router	HASH	['topic-router']	[]	0	
Index Fields:					
Name	Attribute	Type	Field Option	Option Value	
Field Option	Option Value	Field Option	Option Value	Field	
Option	Option Value				
reference_id	reference_id	TAG	SEPARATOR	,	
route_name	route_name	TAG	SEPARATOR	,	

(continues on next page)

(continued from previous page)

↪					↪
reference	reference	TEXT	WEIGHT	1	
↪					
↪					
vector	vector	VECTOR	algorithm	FLAT	↪
↪ data_type	FLOAT32	dim	768	distance_	
↪ metric COSINE					

```
router._index.info()["num_docs"]
```

```
11
```

Simple routing

```
# Query the router with a statement
route_match = router("Can you tell me about the latest in artificial intelligence?")
route_match
```

```
RouteMatch(name='technology', distance=0.419146001339)
```

```
# Query the router with a statement and return a miss
route_match = router("are aliens real?")
route_match
```

```
RouteMatch(name=None, distance=None)
```

We can also route a statement to many routes and order them by distance:

```
# Perform multi-class classification with route_many() -- toggle the max_k and the_
↪ distance_threshold
route_matches = router.route_many("How is AI used in basketball?", max_k=3)
route_matches
```

```
[RouteMatch(name='technology', distance=0.556494116783),
 RouteMatch(name='sports', distance=0.671060025692)]
```

```
# Toggle the aggregation method -- note the different distances in the result
from redisvl.extensions.router.schema import DistanceAggregationMethod

route_matches = router.route_many("How is AI used in basketball?", aggregation_
↪ method=DistanceAggregationMethod.min, max_k=3)
route_matches
```

```
[RouteMatch(name='technology', distance=0.556494116783),
 RouteMatch(name='sports', distance=0.629264295101)]
```

Note the different route match distances. This is because we used the min aggregation method instead of the default avg approach.

Update the routing config

```
from redisvl.extensions.router import RoutingConfig

router.update_routing_config(
    RoutingConfig(aggregation_method=DistanceAggregationMethod.min, max_k=3)
)
```

```
route_matches = router.route_many("Lebron James")
route_matches
```

```
[RouteMatch(name='sports', distance=0.663253962994)]
```

Router serialization

```
router.to_dict()
```

```
{'name': 'topic-router',
 'routes': [{'name': 'technology',
               'references': ['what are the latest advancements in AI?',
                             'tell me about the newest gadgets',
                             "what's trending in tech?"],
               'metadata': {'category': 'tech', 'priority': 1},
               'distance_threshold': 0.71},
            {'name': 'sports',
               'references': ['who won the game last night?',
                             'tell me about the upcoming sports events',
                             "what's the latest in the world of sports?",
                             'sports',
                             'basketball and football'],
               'metadata': {'category': 'sports', 'priority': 2},
               'distance_threshold': 0.72},
            {'name': 'entertainment',
               'references': ['what are the top movies right now?',
                             'who won the best actor award?',
                             "what's new in the entertainment industry?"],
               'metadata': {'category': 'entertainment', 'priority': 3},
               'distance_threshold': 0.7}],
 'vectorizer': {'type': 'hf',
                 'model': 'sentence-transformers/all-mpnet-base-v2'},
 'routing_config': {'max_k': 3, 'aggregation_method': 'min'}}
```

```
router2 = SemanticRouter.from_dict(router.to_dict(), redis_url="redis://localhost:6379")

assert router2.to_dict() == router.to_dict()
```

```
router.to_yaml("router.yaml", overwrite=True)
```

```
router3 = SemanticRouter.from_yaml("router.yaml", redis_url="redis://localhost:6379")

assert router3.to_dict() == router2.to_dict() == router.to_dict()
```

Add route references

```
router.add_route_references(route_name="technology", references=["latest AI trends",
↪ "new tech gadgets"])
```

```
['topic-
↪router:technology:f243fb2d073774e81c7815247cb3013794e6225df3cbe3769cee8c6cefaca777',
 'topic-
↪router:technology:7e4bca5853c1c3298b4d001de13c3c7a79a6e0f134f81acc2e7cddb6845961f']
```

Get route references

```
# by route name
refs = router.get_route_references(route_name="technology")
refs
```

```
[{'id': 'topic-
↪router:technology:f243fb2d073774e81c7815247cb3013794e6225df3cbe3769cee8c6cefaca777',
 'reference_id': 'f243fb2d073774e81c7815247cb3013794e6225df3cbe3769cee8c6cefaca777',
 'route_name': 'technology',
 'reference': 'latest AI trends'},
 {'id': 'topic-
↪router:technology:851f51cce5a9ccfbbcb66993908be6b7871479af3e3a4b139ad292a1bf7e0676',
 'reference_id': '851f51cce5a9ccfbbcb66993908be6b7871479af3e3a4b139ad292a1bf7e0676',
 'route_name': 'technology',
 'reference': 'what are the latest advancements in AI?'},
 {'id': 'topic-
↪router:technology:7e4bca5853c1c3298b4d001de13c3c7a79a6e0f134f81acc2e7cddb6845961f',
 'reference_id': '7e4bca5853c1c3298b4d001de13c3c7a79a6e0f134f81acc2e7cddb6845961f',
 'route_name': 'technology',
 'reference': 'new tech gadgets'},
 {'id': 'topic-
↪router:technology:149a9c9919c58534aa0f369e85ad95ba7f00aa0513e0f81e2aff2ea4a717b0e0',
 'reference_id': '149a9c9919c58534aa0f369e85ad95ba7f00aa0513e0f81e2aff2ea4a717b0e0',
 'route_name': 'technology',
 'reference': 'what's trending in tech?'},
 {'id': 'topic-
↪router:technology:85cc73a1437df27caa2f075a29c497e5a2e532023fbb75378aedbae80779ab37',
 'reference_id': '85cc73a1437df27caa2f075a29c497e5a2e532023fbb75378aedbae80779ab37',
 'route_name': 'technology',
 'reference': 'tell me about the newest gadgets'}]
```

```
# by reference id
refs = router.get_route_references(reference_ids=[refs[0]["reference_id"]])
refs
```

```
[{'id': 'topic-
→router:technology:f243fb2d073774e81c7815247cb3013794e6225df3cbe3769cee8c6cefaca777',
  'reference_id': 'f243fb2d073774e81c7815247cb3013794e6225df3cbe3769cee8c6cefaca777',
  'route_name': 'technology',
  'reference': 'latest AI trends'}]
```

Delete route references

```
# by route name
deleted_count = router.delete_route_references(route_name="sports")
deleted_count
```

```
5
```

```
# by id
deleted_count = router.delete_route_references(reference_ids=[refs[0]["reference_id"]])
deleted_count
```

```
1
```

Clean up the router

```
# Use clear to flush all routes from the index
router.clear()
```

```
# Use delete to clear the index and remove it completely
router.delete()
```

Next Steps

Now that you understand semantic routing, explore these related guides:

- [Manage LLM Message History](#) - Store and retrieve conversation history
- [Cache LLM Responses](#) - Reduce API costs with semantic caching
- [Query and Filter Data](#) - Learn more about filter expressions

Query and Filter Data

This guide covers the filtering capabilities in RedisVL, including tag, numeric, text, geo, and timestamp filters. You'll learn how to combine filters to create complex queries that narrow down search results precisely.

Prerequisites

Before you begin, ensure you have:

- Installed RedisVL: `pip install redisvl`
- A running Redis instance ([Redis 8+](#) or [Redis Cloud](#))

What You'll Learn

By the end of this guide, you will be able to:

- Create and apply tag, numeric, text, and geo filters
- Combine multiple filters using AND/OR logic
- Use `FilterQuery` for non-vector searches
- Execute `CountQuery` to get matching record counts
- Apply `RangeQuery` for distance-based vector searches

```
import pickle
from jupyterutils import table_print, result_print

# load in the example data and printing utils
data = pickle.load(open("hybrid_example_data.pkl", "rb"))
table_print(data)
```

```
<IPython.core.display.HTML object>
```

```
schema = {
    "index": {
        "name": "user_queries",
        "prefix": "user_queries_docs",
        "storage_type": "hash", # default setting -- HASH
    },
    "fields": [
        {"name": "user", "type": "tag"},
        {"name": "credit_score", "type": "tag"},
        {"name": "job", "type": "text"},
        {"name": "age", "type": "numeric"},
        {"name": "last_updated", "type": "numeric"},
        {"name": "office_location", "type": "geo"},
        {
            "name": "user_embedding",
            "type": "vector",
            "attrs": {
                "dims": 3,
                "distance_metric": "cosine",
                "algorithm": "flat",
                "datatype": "float32"
            }
        }
    ]
}
```

(continues on next page)

(continued from previous page)

```
],
}
```

```
from redisvl.index import SearchIndex

# construct a search index from the schema
index = SearchIndex.from_dict(schema, redis_url="redis://localhost:6379")

# create the index (no data yet)
index.create(overwrite=True)
```

```
# load data to redis
keys = index.load(data)
```

```
index.info()['num_docs']
```

```
7
```

Complex Filtering

Complex filtering allows you to combine multiple types of filters in your queries. For example, you may want to search for a user that is a certain age, has a certain job, and is within a certain distance of a location. This is a complex filtering query that combines numeric, tag, and geographic filters.

Tag Filters

Tag filters are filters that are applied to tag fields. These are fields that are not tokenized and are used to store a single categorical value.

```
from redisvl.query import VectorQuery
from redisvl.query.filter import Tag

t = Tag("credit_score") == "high"

v = VectorQuery(
    vector=[0.1, 0.1, 0.5],
    vector_field_name="user_embedding",
    return_fields=["user", "credit_score", "age", "job", "office_location", "last_updated",
↪],
    filter_expression=t
)

results = index.query(v)
result_print(results)
str(v)
```

```
<IPython.core.display.HTML object>
```

```
'(@credit_score:{high})=>[KNN 10 @user_embedding $vector AS vector_distance] RETURN 7,
↪user credit_score age job office_location last_updated vector_distance SORTBY vector_
↪distance ASC DIALECT 2 LIMIT 0 10'
```

```
v.query_string()
```

```
'(@credit_score:{high})=>[KNN 10 @user_embedding $vector AS vector_distance]'
```

```
# negation
t = Tag("credit_score") != "high"

v.set_filter(t)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Performance Tip: For HNSW and SVS-VAMANA indexes, you can add runtime parameters to tune search performance:

```
# Example with runtime parameters for better recall
v = VectorQuery(
    vector=[0.1, 0.1, 0.5],
    vector_field_name="user_embedding",
    return_fields=["user", "credit_score", "age"],
    filter_expression=t,
    ef_runtime=100, # HNSW: higher for better recall
    search_window_size=40 # SVS-VAMANA: larger window for better recall
)
```

These parameters can be adjusted at query time without rebuilding the index. See the *Advanced Queries guide* for more details.

```
# use multiple tags as a list
t = Tag("credit_score") == ["high", "medium"]

v.set_filter(t)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# use multiple tags as a set (to enforce uniqueness)
t = Tag("credit_score") == set(["high", "high", "medium"])

v.set_filter(t)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

What about scenarios where you might want to dynamically generate a list of tags? Have no fear. RedisVL allows you to do this gracefully without having to check for the **empty case**. The **empty case** is when you attempt to run a Tag filter on a field with no defined values to match:

```
Tag("credit_score") == []
```

An empty filter like the one above will yield a * Redis query filter which implies the base case – there is no filter here to use.

```
# gracefully fallback to "*" filter if empty case
empty_case = Tag("credit_score") == []

v.set_filter(empty_case)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Numeric Filters

Numeric filters are filters that are applied to numeric fields and can be used to isolate a range of values for a given field.

```
from redisvl.query.filter import Num

numeric_filter = Num("age").between(15, 35)

v.set_filter(numeric_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# exact match query
numeric_filter = Num("age") == 14

v.set_filter(numeric_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# negation
numeric_filter = Num("age") != 14

v.set_filter(numeric_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Timestamp Filters

In redis all times are stored as an epoch time numeric however, this class allows you to filter with python datetime for ease of use.

```
from redisvl.query.filter import Timestamp
from datetime import datetime

dt = datetime(2025, 3, 16, 13, 45, 39, 132589)
```

(continues on next page)

(continued from previous page)

```
print(f'Epoch comparison: {dt.timestamp()}')

timestamp_filter = Timestamp("last_updated") > dt

v.set_filter(timestamp_filter)
result_print(index.query(v))
```

Epoch comparison: 1742129139.132589

<IPython.core.display.HTML object>

```
from redisvl.query.filter import Timestamp
from datetime import datetime

dt = datetime(2025, 3, 16, 13, 45, 39, 132589)

print(f'Epoch comparison: {dt.timestamp()}')

timestamp_filter = Timestamp("last_updated") < dt

v.set_filter(timestamp_filter)
result_print(index.query(v))
```

Epoch comparison: 1742129139.132589

<IPython.core.display.HTML object>

```
from redisvl.query.filter import Timestamp
from datetime import datetime

dt_1 = datetime(2025, 1, 14, 13, 45, 39, 132589)
dt_2 = datetime(2025, 3, 16, 13, 45, 39, 132589)

print(f'Epoch between: {dt_1.timestamp()} - {dt_2.timestamp()}')

timestamp_filter = Timestamp("last_updated").between(dt_1, dt_2)

v.set_filter(timestamp_filter)
result_print(index.query(v))
```

Epoch between: 1736858739.132589 - 1742129139.132589

<IPython.core.display.HTML object>

Text Filters

Text filters are filters that are applied to text fields. These filters are applied to the entire text field. For example, if you have a text field that contains the text “The quick brown fox jumps over the lazy dog”, a text filter of “quick” will match this text field.

A value passed to `==` or `!=` is matched as a literal phrase, so a `"` or a `\` inside it is insignificant rather than interpreted. The `%` operator is the opposite: its value is a raw pattern, so pass it only patterns your own code composes.

```
from redisvl.query.filter import Text

# exact match filter -- document must contain the exact word doctor
text_filter = Text("job") == "doctor"

v.set_filter(text_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# negation -- document must not contain the exact word doctor
negate_text_filter = Text("job") != "doctor"

v.set_filter(negate_text_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# wildcard match filter
wildcard_filter = Text("job") % "doct*"

v.set_filter(wildcard_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# fuzzy match filter
fuzzy_match = Text("job") % "%%engine%%"

v.set_filter(fuzzy_match)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# conditional -- match documents with job field containing engineer OR doctor
conditional = Text("job") % "engineer|doctor"

v.set_filter(conditional)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# gracefully fallback to "*" filter if empty case
empty_case = Text("job") % ""

v.set_filter(empty_case)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Use raw query strings as input. Below we use the ~ flag to indicate that the full text query is optional. We also choose the BM25 scorer and return document scores along with the result.

```
v.set_filter("(~(@job:engineer))")
v.scorer("BM25").with_scores()

index.query(v)
```

```
[{'id': 'user_queries_docs:01KHKHSW68SH7A1AT1RDG5FC1A',
  'score': 1.8181817787737895,
  'vector_distance': '0',
  'user': 'john',
  'credit_score': 'high',
  'age': '18',
  'job': 'engineer',
  'office_location': '-122.4194,37.7749',
  'last_updated': '1741627789'},
 {'id': 'user_queries_docs:01KHKHSW68SH7A1AT1RDG5FC1B',
  'score': 0.0,
  'vector_distance': '0',
  'user': 'derrick',
  'credit_score': 'low',
  'age': '14',
  'job': 'doctor',
  'office_location': '-122.4194,37.7749',
  'last_updated': '1741627789'},
 {'id': 'user_queries_docs:01KHKHSW68SH7A1AT1RDG5FC1D',
  'score': 1.8181817787737895,
  'vector_distance': '0.109129190445',
  'user': 'tyler',
  'credit_score': 'high',
  'age': '100',
  'job': 'engineer',
  'office_location': '-122.0839,37.3861',
  'last_updated': '1742232589'},
 {'id': 'user_queries_docs:01KHKHSW69W3GWRFZADZM7XMYV',
  'score': 0.0,
  'vector_distance': '0.158808887005',
  'user': 'tim',
  'credit_score': 'high',
  'age': '12',
  'job': 'dermatologist',
  'office_location': '-122.0839,37.3861',
  'last_updated': '1739644189'},
```

(continues on next page)

(continued from previous page)

```
{'id': 'user_queries_docs:01KHKHSW69W3GWRFZADZM7XMYW',
  'score': 0.0,
  'vector_distance': '0.217881977558',
  'user': 'taimur',
  'credit_score': 'low',
  'age': '15',
  'job': 'CEO',
  'office_location': '-122.0839,37.3861',
  'last_updated': '1742232589'},
{'id': 'user_queries_docs:01KHKHSW68SH7A1AT1RDG5FC1C',
  'score': 0.0,
  'vector_distance': '0.266666650772',
  'user': 'nancy',
  'credit_score': 'high',
  'age': '94',
  'job': 'doctor',
  'office_location': '-122.4194,37.7749',
  'last_updated': '1710696589'},
{'id': 'user_queries_docs:01KHKHSW69W3GWRFZADZM7XMYX',
  'score': 0.0,
  'vector_distance': '0.653301358223',
  'user': 'joe',
  'credit_score': 'medium',
  'age': '35',
  'job': 'dentist',
  'office_location': '-122.0839,37.3861',
  'last_updated': '1742232589'}]
```

Geographic Filters

Geographic filters are filters that are applied to geographic fields. These filters are used to find results that are within a certain distance of a given point. The distance is specified in kilometers, miles, meters, or feet. A radius can also be specified to find results within a certain radius of a given point.

```
from redisvl.query.filter import Geo, GeoRadius

# within 10 km of San Francisco office
geo_filter = Geo("office_location") == GeoRadius(-122.4194, 37.7749, 10, "km")

v.set_filter(geo_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# within 100 km Radius of San Francisco office
geo_filter = Geo("office_location") == GeoRadius(-122.4194, 37.7749, 100, "km")

v.set_filter(geo_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# not within 10 km Radius of San Francisco office
geo_filter = Geo("office_location") != GeoRadius(-122.4194, 37.7749, 10, "km")

v.set_filter(geo_filter)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Combining Filters

This example combines a numeric filter with a tag filter to search for users between ages 20 and 30 who have the job “engineer”.

Intersection (“and”)

```
t = Tag("credit_score") == "high"
low = Num("age") >= 18
high = Num("age") <= 100
ts = Timestamp("last_updated") > datetime(2025, 3, 16, 13, 45, 39, 132589)

combined = t & low & high & ts

v = VectorQuery([0.1, 0.1, 0.5],
                "user_embedding",
                return_fields=["user", "credit_score", "age", "job", "office_location"],
                filter_expression=combined)

result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Union (“or”)

The union of two queries is the set of all results that are returned by either of the two queries. The union of two queries is performed using the `|` operator.

```
low = Num("age") < 18
high = Num("age") > 93

combined = low | high

v.set_filter(combined)
result_print(index.query(v))
```



```
<IPython.core.display.HTML object>
```

Dynamic Combination

There are often situations where you may or may not want to use a filter in a given query. As shown above, filters will except the None type and revert to a wildcard filter essentially returning all results.

The same goes for filter combinations which enables rapid reuse of filters in requests with different parameters as shown below. This removes the need for a number of “if-then” conditionals to test for the empty case.

```
def make_filter(age=None, credit=None, job=None):
    flexible_filter = (
        (Num("age") > age) &
        (Tag("credit_score") == credit) &
        (Text("job") % job)
    )
    return flexible_filter
```

```
# all parameters
combined = make_filter(age=18, credit="high", job="engineer")
v.set_filter(combined)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# just age and credit_score
combined = make_filter(age=18, credit="high")
v.set_filter(combined)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# just age
combined = make_filter(age=18)
v.set_filter(combined)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

```
# no filters
combined = make_filter()
v.set_filter(combined)
result_print(index.query(v))
```

```
<IPython.core.display.HTML object>
```

Non-vector Queries

When you need to query without vector similarity (similar to a SQL WHERE clause), use the `FilterQuery` class. It accepts a `FilterExpression` and returns matching records.

```
from redisvl.query import FilterQuery

has_low_credit = Tag("credit_score") == "low"

filter_query = FilterQuery(
    return_fields=["user", "credit_score", "age", "job", "location"],
    filter_expression=has_low_credit
)

results = index.query(filter_query)

result_print(results)
```

```
<IPython.core.display.HTML object>
```

Count Queries

Use `CountQuery` with a `FilterExpression` to get the count of matching records without retrieving the actual data.

```
from redisvl.query import CountQuery

has_low_credit = Tag("credit_score") == "low"

filter_query = CountQuery(filter_expression=has_low_credit)

count = index.query(filter_query)

print(f"{count} records match the filter expression {str(has_low_credit)} for the given_
↪index.")
```

```
2 records match the filter expression @credit_score:{low} for the given index.
```

Range Queries

Range Queries are a useful method to perform a vector search where only results within a vector `distance_threshold` are returned. This enables the user to find all records within their dataset that are similar to a query vector where “similar” is defined by a quantitative value.

```
from redisvl.query import RangeQuery

range_query = RangeQuery(
    vector=[0.1, 0.1, 0.5],
    vector_field_name="user_embedding",
    return_fields=["user", "credit_score", "age", "job", "location"],
    distance_threshold=0.2
```

(continues on next page)

(continued from previous page)

```
)

# same as the vector query or filter query
results = index.query(range_query)

result_print(results)
```

```
<IPython.core.display.HTML object>
```

The distance threshold can be changed between queries. Setting `distance_threshold=0.1` returns only matches within 0.1 distance of the query vector, resulting in fewer but more similar matches.

```
range_query.set_distance_threshold(0.1)

result_print(index.query(range_query))
```

```
<IPython.core.display.HTML object>
```

Range queries can also be used with filters like any other query type. The following limits the results to only include records with a job of engineer while also being within the vector range (aka distance).

```
is_engineer = Text("job") == "engineer"

range_query.set_filter(is_engineer)

result_print(index.query(range_query))
```

```
<IPython.core.display.HTML object>
```

Advanced Query Modifiers

See all modifier options available on the query API docs: <https://docs.redisvl.com/en/latest/api/query.html>

```
# Sort by a different field and change dialect
v = VectorQuery(
    vector=[0.1, 0.1, 0.5],
    vector_field_name="user_embedding",
    return_fields=["user", "credit_score", "age", "job", "office_location"],
    num_results=5,
    filter_expression=is_engineer
).sort_by("age", asc=False).dialect(3)

result = index.query(v)
result_print(result)
```

```
<IPython.core.display.HTML object>
```

Raw Redis Query String

Sometimes it's helpful to convert these classes into their raw Redis query strings.

```
# check out the complex query from above
str(v)
```

```
'(@job:("engineer"))=>[KNN 5 @user_embedding $vector AS vector_distance] RETURN 6 user_
↪credit_score age job office_location vector_distance SORTBY age DESC DIALECT 3 LIMIT 0_
↪5'
```

```
t = Tag("credit_score") == "high"

str(t)
```

```
'@credit_score:{high}'
```

```
t = Tag("credit_score") == "high"
low = Num("age") >= 18
high = Num("age") <= 100

combined = t & low & high

str(combined)
```

```
'((@credit_score:{high} @age:[18 +inf]) @age:[-inf 100])'
```

The RedisVL SearchIndex class exposes a search() method which is a simple wrapper around the FT.SEARCH API. Provide any valid Redis query string.

```
results = index.search(str(t))
for r in results.docs:
    print(r.__dict__)
```

```
{'id': 'user_queries_docs:01KHKHSW68SH7A1AT1RDG5FC1A', 'payload': None, 'user': 'john',
↪ 'age': '18', 'job': 'engineer', 'credit_score': 'high', 'office_location': '-122.4194,
↪ 37.7749', 'user_embedding': '==\x00\x00\x00?', 'last_updated': '1741627789'}
{'id': 'user_queries_docs:01KHKHSW68SH7A1AT1RDG5FC1C', 'payload': None, 'user': 'nancy',
↪ 'age': '94', 'job': 'doctor', 'credit_score': 'high', 'office_location': '-122.4194,37.
↪ 7749', 'user_embedding': '333?=\x00\x00\x00?', 'last_updated': '1710696589'}
{'id': 'user_queries_docs:01KHKHSW68SH7A1AT1RDG5FC1D', 'payload': None, 'user': 'tyler',
↪ 'age': '100', 'job': 'engineer', 'credit_score': 'high', 'office_location': '-122.0839,
↪ 37.3861', 'user_embedding': '=>\x00\x00\x00?', 'last_updated': '1742232589'}
{'id': 'user_queries_docs:01KHKHSW69W3GWRFZADZM7XMYV', 'payload': None, 'user': 'tim',
↪ 'age': '12', 'job': 'dermatologist', 'credit_score': 'high', 'office_location': '-122.
↪ 0839,37.3861', 'user_embedding': '>>\x00\x00\x00?', 'last_updated': '1739644189'}
```

Next Steps

Now that you understand filtering in RedisVL, explore these related guides:

- [Use Advanced Query Types](#) - Learn about TextQuery, HybridQuery, and MultiVectorQuery
- [Cache LLM Responses](#) - Use filters with semantic caching for multi-user scenarios
- [Write SQL Queries for Redis](#) - Use SQL-like syntax for Redis queries

Cleanup

```
index.delete()
```

Create Embeddings with Vectorizers

This guide demonstrates how to create embeddings using RedisVL's built-in text vectorizers. RedisVL supports multiple embedding providers: OpenAI, HuggingFace, Ollama, Google (Vertex AI and the Gemini Developer API), Cohere, Mistral AI, Amazon Bedrock, VoyageAI, and custom vectorizers.

Prerequisites

Before you begin, ensure you have:

- Installed RedisVL: `pip install redisvl`
- A running Redis instance ([Redis 8+](#) or [Redis Cloud](#))
- API keys or local model servers for the embedding providers you plan to use

What You'll Learn

By the end of this guide, you will be able to:

- Create embeddings using multiple providers (OpenAI, HuggingFace, Ollama, Cohere, etc.)
- Use synchronous and asynchronous embedding methods
- Batch embed multiple texts efficiently
- Build custom vectorizers for your own embedding functions
- Integrate vectorizers with RedisVL indexes for semantic search

```
# import necessary modules
import os
```

Creating Text Embeddings

This example will show how to create an embedding from 3 simple sentences with a number of different text vectorizers in RedisVL.

- “That is a happy dog”
- “That is a happy person”
- “Today is a nice day”

OpenAI

The `OpenAITextVectorizer` makes it simple to use RedisVL with the embeddings models at OpenAI. For this you will need to install `openai`.

```
pip install openai
```

```
import getpass

# setup the API Key
api_key = os.environ.get("OPENAI_API_KEY") or getpass.getpass("Enter your OpenAI API_
↪key: ")
```

```
from redisvl.utils.vectorize import OpenAITextVectorizer

# create a vectorizer
oai = OpenAITextVectorizer(
    model="text-embedding-ada-002",
    api_config={"api_key": api_key},
)

test = oai.embed("This is a test sentence.")
print("Vector dimensions: ", len(test))
test[:10]
```

```
Vector dimensions: 1536
```

```
[-0.0011391325388103724,
 -0.003206387162208557,
 0.002380132209509611,
 -0.004501554183661938,
 -0.010328996926546097,
 0.012922565452754498,
 -0.005491119809448719,
 -0.0029864837415516376,
 -0.007327961269766092,
 -0.03365817293524742]
```

```
# Create many embeddings at once
sentences = [
    "That is a happy dog",
```

(continues on next page)

(continued from previous page)

```

    "That is a happy person",
    "Today is a sunny day"
]

embeddings = oai.embed_many(sentences)
embeddings[0][:10]

```

```

[-0.017466850578784943,
 1.8471690054866485e-05,
 0.00129731057677418,
 -0.02555876597762108,
 -0.019842341542243958,
 0.01603139191865921,
 -0.0037347301840782166,
 0.0009670283179730177,
 0.006618348415941,
 -0.02497442066669464]

```

```

# openai also supports asynchronous requests, which we can use to speed up the
↪ vectorization process.
embeddings = await oai.aembed_many(sentences)
print("Number of Embeddings:", len(embeddings))

```

```

Number of Embeddings: 3

```

Azure OpenAI

The `AzureOpenAITextVectorizer` is a variation of the OpenAI vectorizer that calls OpenAI models within Azure. If you've already installed `openai`, then you're ready to use Azure OpenAI.

The only practical difference between OpenAI and Azure OpenAI is the variables required to call the API.

```

# NBVAL_SKIP
# additionally to the API Key, setup the API endpoint and version
api_key = os.environ.get("AZURE_OPENAI_API_KEY") or getpass.getpass("Enter your
↪ AzureOpenAI API key: ")
api_version = os.environ.get("OPENAI_API_VERSION") or getpass.getpass("Enter your
↪ AzureOpenAI API version: ")
azure_endpoint = os.environ.get("AZURE_OPENAI_ENDPOINT") or getpass.getpass("Enter your
↪ AzureOpenAI API endpoint: ")
deployment_name = os.environ.get("AZURE_OPENAI_DEPLOYMENT_NAME", "text-embedding-ada-002
↪ ")

# Skip Azure examples when required values are missing (e.g. Run All without Azure
↪ configured).
_azure_configured = bool(azure_endpoint and api_key and api_version)

```

```

# NBVAL_SKIP
# Depends on the Azure OpenAI cell above, which is not executed in CI.
from redisvl.utils.vectorize import AzureOpenAITextVectorizer

```

(continues on next page)

(continued from previous page)

```

if not _azure_configured:
    print("Skipping Azure OpenAI example: set AZURE_OPENAI_ENDPOINT, AZURE_OPENAI_API_
    ↪KEY and OPENAI_API_VERSION")
    az_oai = None
else:
    az_oai = AzureOpenAITextVectorizer(
        model=deployment_name, # Must be your CUSTOM deployment name
        api_config={
            "api_key": api_key,
            "api_version": api_version,
            "azure_endpoint": azure_endpoint,
        },
    )

    test = az_oai.embed("This is a test sentence.")
    print("Vector dimensions: ", len(test))
    test[:10]

```

```

# NBVAL_SKIP
# Depends on the Azure OpenAI cell above, which is not executed in CI.
# Just like OpenAI, AzureOpenAI supports batching embeddings and asynchronous requests.
sentences = [
    "That is a happy dog",
    "That is a happy person",
    "Today is a sunny day",
]

if _azure_configured and az_oai is not None:
    embeddings = await az_oai.aembed_many(sentences)
    embeddings[0][:10]
else:
    print("Skipping: run the Azure cells above with valid configuration.")

```

Huggingface

Huggingface is a popular NLP platform that has a number of pre-trained models you can use off the shelf. RedisVL supports using Huggingface “Sentence Transformers” to create embeddings from text. To use Huggingface, you will need to install the sentence-transformers library.

```
pip install sentence-transformers
```

```

os.environ["TOKENIZERS_PARALLELISM"] = "false"
from redisvl.utils.vectorize import HFTextVectorizer

# create a vectorizer
# choose your model from the huggingface website
hf = HFTextVectorizer(model="sentence-transformers/all-mpnet-base-v2")

```

(continues on next page)

(continued from previous page)

```
# embed a sentence
test = hf.embed("This is a test sentence.")
test[:10]
```

```
# You can also create many embeddings at once
embeddings = hf.embed_many(sentences, as_buffer=True)
```

Ollama

Ollama lets you run embedding models locally. RedisVL supports Ollama through the `OllamaTextVectorizer`.

Install the Python client and pull an embedding model before running this example:

```
pip install 'redisvl[ollama]'
ollama pull nomic-embed-text
```

Make sure the Ollama daemon is running with `ollama serve`. By default, the Ollama client uses `OLLAMA_HOST` if set, otherwise it connects to `http://localhost:11434`. To connect to a custom Ollama server explicitly, pass `host="http://your-host:11434"` when creating the vectorizer.

```
# NBVAL_SKIP
# No Ollama server in CI, so this only burns time on connection retries.
from redisvl.utils.vectorize import OllamaTextVectorizer

ollama_model = os.environ.get("OLLAMA_MODEL", "nomic-embed-text")

try:
    ollama = OllamaTextVectorizer(model=ollama_model)

    test = ollama.embed("This is a test sentence.")
    print("Vector dimensions:", len(test))
    print(test[:10])
except (ImportError, ConnectionError, ValueError) as exc:
    print("Skipping Ollama example:", exc)
    ollama = None
```

```
# NBVAL_SKIP
# Depends on the Ollama cell above, which is not executed in CI.
if ollama is not None:
    embeddings = ollama.embed_many(sentences, batch_size=2)
    print("Number of embeddings:", len(embeddings))
    print("Vector dimensions:", len(embeddings[0]))
else:
    print("Skipping: run the Ollama cell above with a running Ollama server and pulled_
    ↪model.")
```

```
# NBVAL_SKIP
# Depends on the Ollama cell above, which is not executed in CI.
if ollama is not None:
    embeddings = await ollama.aembed_many(sentences, batch_size=2)
    print("Number of async embeddings:", len(embeddings))
```

(continues on next page)

(continued from previous page)

```
else:
    print("Skipping: run the Ollama cell above with a running Ollama server and pulled_
↪model.")
```

Deprecated. VertexAIVectorizer uses Google's Vertex AI model-garden SDK, which Google has deprecated with a scheduled removal. For text embeddings use the **Google Gen AI** vectorizer shown in the next section (GoogleGenAIVectorizer) on the supported google-genai SDK. Multimodal (image/video) migration is tracked in [issue #620](#).

VertexAI

VertexAI is GCP's fully-featured AI platform including a number of pretrained LLMs. RedisVL supports using VertexAI to create embeddings from these models. To use VertexAI, you will first need to install the google-cloud-aiplatform library.

```
pip install google-cloud-aiplatform>=1.26
```

1. Then you need to gain access to a [Google Cloud Project](#) and provide [access to credentials](#). This is accomplished by setting the GOOGLE_APPLICATION_CREDENTIALS environment variable pointing to the path of a JSON key file downloaded from your service account on GCP.
2. Lastly, you need to find your [project ID](#) and [geographic region for VertexAI](#).

Make sure the following env vars are set:

```
GOOGLE_APPLICATION_CREDENTIALS=<path to your gcp JSON creds>
GCP_PROJECT_ID=<your gcp project id>
GCP_LOCATION=<your gcp geo region for vertex ai>
```

```
# NBVAL_SKIP
# Deprecated vectorizer; not executed in CI so notebook validation makes no API calls.
from redisvl.utils.vectorize import VertexAIVectorizer

# create a vectorizer
vtx = VertexAIVectorizer(api_config={
    "project_id": os.environ.get("GCP_PROJECT_ID") or getpass.getpass("Enter your GCP_
↪Project ID: "),
    "location": os.environ.get("GCP_LOCATION") or getpass.getpass("Enter your GCP_
↪Location: "),
    "google_application_credentials": os.environ.get("GOOGLE_APPLICATION_CREDENTIALS")_
↪or getpass.getpass("Enter your Google App Credentials path: ")
})

# embed a sentence
test = vtx.embed("This is a test sentence.")
test[:10]
```

Google Gen AI

The GoogleGenAIVectorizer uses Google's `google-genai` SDK — the supported replacement for the deprecated Vertex AI model-garden SDK. It reaches **both** Google embedding backends from a single client:

- **Vertex AI / Gemini Enterprise** — GCP project auth (project_id + location, credentials via ADC).
- **Gemini Developer API** — a single api_key.

Install it with `pip install redisvl[google-genai]`.

Migrating from VertexAIVectorizer:

	VertexAIVectorizer (deprecated)	GoogleGenAIVectorizer
SDK	google-cloud-aiplatform (deprecated)	google-genai (supported)
Default model	textembedding-gecko (768 dims)	gemini-embedding-001 (3072 dims)
Backends	Vertex AI only	Vertex AI and Gemini Developer API
Async	no	yes (aembed, aembed_many)

Because the default model and dimensions differ, embeddings are **not** interchangeable — reindex when you switch. When you request a reduced output_dimensionality, the vectorizer L2-normalizes the result so it stays valid for Redis COSINE / inner-product search.

Set one of the following:

```
# Gemini Developer API
GEMINI_API_KEY=<your gemini api key>

# or Vertex AI
GOOGLE_CLOUD_PROJECT=<your gcp project id>
GOOGLE_CLOUD_LOCATION=<your gcp region>
GOOGLE_APPLICATION_CREDENTIALS=<path to your gcp JSON creds>
```

```
# NBVAL_SKIP
# Docs example; not executed in CI so notebook validation makes no API calls.
from redisvl.utils.vectorize import GoogleGenAIVectorizer

# Auto-detects the backend: GEMINI_API_KEY -> Gemini, else GCP project/location ->
# ↪ Vertex AI.
# Guarded so this notebook still runs when Google credentials aren't configured.
try:
    genai_vectorizer = GoogleGenAIVectorizer(model="gemini-embedding-001")
    print(f"backend={genai_vectorizer.backend}, dims={genai_vectorizer.dims}")
    genai_test = genai_vectorizer.embed("This is a test sentence.")
    print(genai_test[:10])
except (ImportError, ValueError) as e:
    print(f"Skipping GoogleGenAIVectorizer demo: {e}")
    genai_vectorizer = None
```

Cohere

Cohere allows you to implement language AI into your product. The `CohereTextVectorizer` makes it simple to use RedisVL with the embeddings models at Cohere. For this you will need to install `cohere`.

```
pip install cohere
```

```
import getpass
# setup the API Key
api_key = os.environ.get("COHERE_API_KEY") or getpass.getpass("Enter your Cohere API_
↪key: ")
```

Special attention needs to be paid to the `input_type` parameter for each `embed` call. For example, for embedding queries, you should set `input_type='search_query'`; for embedding documents, set `input_type='search_document'`. See more information [here](#)

```
from redisvl.utils.vectorize import CohereTextVectorizer

# create a vectorizer
co = CohereTextVectorizer(
    model="embed-english-v3.0",
    api_config={"api_key": api_key},
)

# embed a search query
test = co.embed("This is a test sentence.", input_type='search_query')
print("Vector dimensions: ", len(test))
print(test[:10])

# embed a document
test = co.embed("This is a test sentence.", input_type='search_document')
print("Vector dimensions: ", len(test))
print(test[:10])
```

Learn more about using RedisVL and Cohere together through [this dedicated user guide](#).

VoyageAI

VoyageAI allows you to implement language AI into your product. The `VoyageAIVectorizer` makes it simple to use RedisVL with the embeddings models at VoyageAI. For this you will need to install `voyageai`.

```
pip install voyageai
```

```
import getpass
# setup the API Key
api_key = os.environ.get("VOYAGE_API_KEY") or getpass.getpass("Enter your VoyageAI API_
↪key: ")
```

Special attention needs to be paid to the `input_type` parameter for each `embed` call. For example, for embedding queries, you should set `input_type='query'`; for embedding documents, set `input_type='document'`. See more information [here](#)

```

from redisvl.utils.vectorize import VoyageAIVectorizer

# create a vectorizer
vo = VoyageAIVectorizer(
    model="voyage-law-2", # Please check the available models at https://docs.voyageai.
    ↪com/docs/embeddings
    api_config={"api_key": api_key},
)

# embed a search query
test = vo.embed("This is a test sentence.", input_type='query')
print("Vector dimensions: ", len(test))
print(test[:10])

# embed a document
test = vo.embed("This is a test sentence.", input_type='document')
print("Vector dimensions: ", len(test))
print(test[:10])

```

Contextualized embeddings

VoyageAI's `voyage-context-*` models support *contextualized* chunk embeddings. The `VoyageAIVectorizer` automatically routes `voyage-context-*` models to the contextualized embeddings API: pass your list of chunk strings to `embed_many` and you get one embedding back per chunk. Each input string is sent as its own auto-chunked document, so the inputs are embedded **independently** (a chunk's embedding does not depend on the other strings in the list). This preserves the one-embedding-per-input contract and keeps the embeddings cache deterministic.

```

# NBVAL_SKIP
# Contextualized embeddings (voyage-context-* models). Each input string is sent
# as its own auto-chunked document and embedded independently, so the vectorizer
# returns exactly one embedding per input string. Requires voyageai>=0.5.0.
context_vo = VoyageAIVectorizer(
    model="voyage-context-4", # See https://docs.voyageai.com/docs/contextualized-chunk-
    ↪embeddings
    api_config={"api_key": api_key},
)

chunks = [
    "That is a happy dog",
    "That is a happy person",
    "Today is a sunny day",
]
context_embeddings = context_vo.embed_many(chunks, input_type="document")
print("Number of embeddings:", len(context_embeddings))
print("Vector dimensions:", len(context_embeddings[0]))

```

Mistral AI

Mistral offers LLM and embedding APIs for you to implement into your product. The `MistralAITextVectorizer` makes it simple to use RedisVL with their embeddings model. You will need to install `mistralai`.

```
pip install mistralai
```

```
from redisvl.utils.vectorize import MistralAITextVectorizer

mistral = MistralAITextVectorizer()

# embed a sentence using their asynchronous method
test = await mistral.aembed("This is a test sentence.")
print("Vector dimensions: ", len(test))
print(test[:10])
```

Amazon Bedrock

Amazon Bedrock provides fully managed foundation models for text embeddings. Install the required dependencies:

```
pip install 'redisvl[bedrock]' # Installs boto3
```

Configure AWS credentials:

```
import os
import getpass

if "AWS_ACCESS_KEY_ID" not in os.environ:
    os.environ["AWS_ACCESS_KEY_ID"] = getpass.getpass("Enter AWS Access Key ID: ")
if "AWS_SECRET_ACCESS_KEY" not in os.environ:
    os.environ["AWS_SECRET_ACCESS_KEY"] = getpass.getpass("Enter AWS Secret Key: ")

os.environ["AWS_REGION"] = "us-east-1" # Change as needed
```

Create embeddings:

```
from redisvl.utils.vectorize import BedrockVectorizer

bedrock = BedrockVectorizer(
    model="amazon.titan-embed-text-v2:0"
)

# Single embedding
text = "This is a test sentence."
embedding = bedrock.embed(text)
print(f"Vector dimensions: {len(embedding)}")

# Multiple embeddings
```

(continues on next page)

(continued from previous page)

```

sentences = [
    "That is a happy dog",
    "That is a happy person",
    "Today is a sunny day"
]
embeddings = bedrock.embed_many(sentences)

```

Custom Vectorizers

RedisVL supports the use of other vectorizers and provides a class to enable compatibility with any function that generates a vector or vectors from string data

```

from redisvl.utils.vectorize import CustomVectorizer

def generate_embeddings(text_input, **kwargs):
    return [0.101] * 768

custom_vectorizer = CustomVectorizer(generate_embeddings)

custom_vectorizer.embed("This is a test sentence.")[:10]

```

This enables the use of custom vectorizers with other RedisVL components

```

from redisvl.extensions.cache.llm import SemanticCache

cache = SemanticCache(name="custom_cache", vectorizer=custom_vectorizer)

cache.store("this is a test prompt", "this is a test response")
cache.check("this is also a test prompt")

```

Search with Provider Embeddings

Now that we've created our embeddings, we can use them to search for similar sentences. We will use the same 3 sentences from above and search for similar sentences.

First, we need to create the schema for our index.

Here's what the schema for the example looks like in yaml for the HuggingFace vectorizer:

```

version: '0.1.0'

index:
  name: vectorizers
  prefix: doc
  storage_type: hash

fields:
  - name: sentence
    type: text
  - name: embedding
    type: vector

```

(continues on next page)

(continued from previous page)

```
attrs:
  dims: 768
  algorithm: flat
  distance_metric: cosine
```

```
from redisvl.index import SearchIndex

# construct a search index from the schema
index = SearchIndex.from_yaml("./schema.yaml", redis_url="redis://localhost:6379")

# create the index (no data yet)
index.create(overwrite=True)
```

```
# use the CLI to see the created index
!rvm index listall
```

Loading data to RedisVL is easy. It expects a list of dictionaries. The vector is stored as bytes.

```
from redisvl.redis.utils import array_to_buffer

embeddings = hf.embed_many(sentences)

data = [{"text": t,
         "embedding": array_to_buffer(v, dtype="float32")}
        for t, v in zip(sentences, embeddings)]

index.load(data)
```

```
from redisvl.query import VectorQuery

# use the HuggingFace vectorizer again to create a query embedding
query_embedding = hf.embed("That is a happy cat")

query = VectorQuery(
    vector=query_embedding,
    vector_field_name="embedding",
    return_fields=["text"],
    num_results=3
)

results = index.query(query)
for doc in results:
    print(doc["text"], doc["vector_distance"])
```


Selecting your float data type

When embedding text as byte arrays RedisVL supports 4 different floating point data types, float16, float32, float64 and bfloat16, and 2 integer types, int8 and uint8. Your dtype set for your vectorizer must match what is defined in your search index. If one is not explicitly set the default is float32.

```
vectorizer = HFTextVectorizer(dtype="float16")

# subsequent calls to embed(", as_buffer=True) and embed_many(", as_buffer=True) will now
↳ encode as float16
float16_bytes = vectorizer.embed('test sentence', as_buffer=True)

# to generate embeddings with different dtype instantiate a new vectorizer
vectorizer_64 = HFTextVectorizer(dtype='float64')
float64_bytes = vectorizer_64.embed('test sentence', as_buffer=True)

float16_bytes != float64_bytes
```

Next Steps

Now that you understand how to create embeddings, explore these related guides:

- *Getting Started* - Learn the basics of indexes and queries
- *Rerank Results* - Improve search quality with reranking models
- *Cache Embeddings* - Cache embedding vectors for faster repeated computations

Cleanup

```
index.delete()
```

Choose a Storage Type

Redis provides a [variety of data structures](#) that can adapt to your domain-specific applications. This guide demonstrates how to use RedisVL with both [Hash](#) and [JSON](#) storage types, helping you choose the right approach for your use case.

Prerequisites

Before you begin, ensure you have:

- Installed RedisVL: `pip install redisvl`
- A running Redis instance ([Redis 8+](#) or [Redis Cloud](#))

What You'll Learn

By the end of this guide, you will be able to:

- Understand the differences between Hash and JSON storage types
- Define schemas for both Hash and JSON storage
- Load and query data using each storage type
- Access nested JSON fields using JSONPath expressions
- Choose the right storage type for your application

```
# import necessary modules
import pickle

from redisvl.redis.utils import buffer_to_array
from redisvl.index import SearchIndex

# load in the example data and printing utils
data = pickle.load(open("hybrid_example_data.pkl", "rb"))
```

```
from jupyterutils import result_print, table_print

table_print(data)
```

```
<IPython.core.display.HTML object>
```

Hash or JSON: How to Choose

Both storage options offer different features and tradeoffs. This section walks through a sample dataset to illustrate when and how to use each option.

Working with Hashes

Hashes in Redis are simple collections of field-value pairs. Think of it like a mutable single-level dictionary contains multiple “rows”:

```
{
  "model": "Deimos",
  "brand": "Ergonom",
  "type": "Enduro bikes",
  "price": 4972,
}
```

Hashes are best suited for use cases with the following characteristics:

- Performance (speed) and storage space (memory consumption) are top concerns
- Data can be easily normalized and modeled as a single-level dict

Hashes are typically the default recommendation.

```
# define the hash index schema
hash_schema = {
    "index": {
        "name": "user-hash",
        "prefix": "user-hash-docs",
        "storage_type": "hash", # default setting -- HASH
    },
    "fields": [
        {"name": "user", "type": "tag"},
        {"name": "credit_score", "type": "tag"},
        {"name": "job", "type": "text"},
        {"name": "age", "type": "numeric"},
        {"name": "office_location", "type": "geo"},
        {
            "name": "user_embedding",
            "type": "vector",
            "attrs": {
                "dims": 3,
                "distance_metric": "cosine",
                "algorithm": "flat",
                "datatype": "float32"
            }
        }
    ]
}
```

```
# construct a search index from the hash schema
hindex = SearchIndex.from_dict(hash_schema, redis_url="redis://localhost:6379")

# create the index (no data yet)
hindex.create(overwrite=True)
```

```
# show the underlying storage type
hindex.storage_type
```

```
<StorageType.HASH: 'hash'>
```

Vectors as byte strings

One nuance when working with Hashes in Redis, is that all vectorized data must be passed as a byte string (for efficient storage, indexing, and processing). An example of that can be seen below:

```
# show a single entry from the data that will be loaded
data[0]
```

```
{'user': 'john',
 'age': 18,
 'job': 'engineer',
 'credit_score': 'high',
```

(continues on next page)

(continued from previous page)

```
'office_location': '-122.4194,37.7749',
'user_embedding': b'\xcd\xcc\xcc=\xcd\xcc\xcc=\x00\x00\x00?',
'last_updated': 1741627789}
```

```
# load hash data
keys = hindex.load(data)
```

```
!rvl stats -i user-hash
```

Statistics:

Stat Key	Value
num_docs	7
num_terms	6
max_doc_id	7
num_records	44
percent_indexed	1
hash_indexing_failures	0
number_of_uses	1
bytes_per_record_avg	39.0681800
doc_table_size_mb	0.00837230
inverted_sz_mb	0.00163936
key_table_size_mb	3.50952148
offset_bits_per_record_avg	8
offset_vectors_sz_mb	8.58306884
offsets_per_term_avg	0.20454545
records_per_doc_avg	6.28571414
sortable_values_size_mb	0
total_indexing_time	0.55204
total_inverted_index_blocks	18
vector_index_sz_mb	0.02820587

Performing Queries

Once our index is created and data is loaded into the right format, we can run queries against the index with RedisVL:

```
from redisvl.query import VectorQuery
from redisvl.query.filter import Tag, Text, Num

t = (Tag("credit_score") == "high") & (Text("job") % "enginee*") & (Num("age") > 17) #_
↳ codespell:ignore enginee

v = VectorQuery(
    vector=[0.1, 0.1, 0.5],
    vector_field_name="user_embedding",
    return_fields=["user", "credit_score", "age", "job", "office_location"],
    filter_expression=t
)
```

(continues on next page)

(continued from previous page)

```
results = hindex.query(v)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

```
# clean up
hindex.delete()
```

Working with JSON

JSON is best suited for use cases with the following characteristics:

- Ease of use and data model flexibility are top concerns
- Application data is already native JSON
- Replacing another document storage/db solution

```
# define the json index schema
json_schema = {
    "index": {
        "name": "user-json",
        "prefix": "user-json-docs",
        "storage_type": "json", # JSON storage type
    },
    "fields": [
        {"name": "user", "type": "tag"},
        {"name": "credit_score", "type": "tag"},
        {"name": "job", "type": "text"},
        {"name": "age", "type": "numeric"},
        {"name": "office_location", "type": "geo"},
        {
            "name": "user_embedding",
            "type": "vector",
            "attrs": {
                "dims": 3,
                "distance_metric": "cosine",
                "algorithm": "flat",
                "datatype": "float32"
            }
        }
    ]
}
```

```
# construct a search index from the json schema
jindex = SearchIndex.from_dict(json_schema, redis_url="redis://localhost:6379")

# create the index (no data yet)
jindex.create(overwrite=True)
```

Vectors as Float Arrays

Vectorized data stored in JSON must be a pure array (Python list) of floats. The following code modifies the sample data to use this format:

```
json_data = data.copy()

for d in json_data:
    d['user_embedding'] = buffer_to_array(d['user_embedding'], dtype='float32')
```

```
# inspect a single JSON record
json_data[0]
```

```
{'user': 'john',
 'age': 18,
 'job': 'engineer',
 'credit_score': 'high',
 'office_location': '-122.4194,37.7749',
 'user_embedding': [0.10000000149011612, 0.10000000149011612, 0.5],
 'last_updated': 1741627789}
```

```
keys = jindex.load(json_data)
```

```
# we can now run the exact same query as above
result_print(jindex.query(v))
```

```
<IPython.core.display.HTML object>
```

Cleanup

```
jindex.delete()
```

Working with nested data in JSON

Redis also supports native **JSON** objects. These can be multi-level (nested) objects, with full JSONPath support for updating/retrieving sub elements:

```
{
  "name": "Specialized Stump jumper",
  "metadata": {
    "model": "Stumpjumper",
    "brand": "Specialized",
    "type": "Enduro bikes",
    "price": 3000
  },
}
```

Full JSON Path support

Because Redis enables full JSON path support, when creating an index schema, elements need to be indexed and selected by their path with the desired name AND path that points to where the data is located within the objects.

By default, RedisVL will assume the path as `$.{name}` if not provided in JSON fields schema. If nested provide path as `$.object.attribute`

As an example:

```
from redisvl.utils.vectorize import HFTextVectorizer

emb_model = HFTextVectorizer()

bike_data = [
    {
        "name": "Specialized Stump jumper",
        "metadata": {
            "model": "Stumpjumper",
            "brand": "Specialized",
            "type": "Enduro bikes",
            "price": 3000
        },
        "description": "The Specialized Stumpjumper is a versatile enduro bike that
↳ dominates both climbs and descents. Features a FACT 11m carbon fiber frame, FOX FLOAT
↳ suspension with 160mm travel, and SRAM X01 Eagle drivetrain. The asymmetric frame
↳ design and internal storage compartment make it a practical choice for all-day
↳ adventures."
    },
    {
        "name": "bike_2",
        "metadata": {
            "model": "Slash",
            "brand": "Trek",
            "type": "Enduro bikes",
            "price": 5000
        },
        "description": "Trek's Slash is built for aggressive enduro riding and racing.
↳ Featuring Trek's Alpha Aluminum frame with RE:aktiv suspension technology, 160mm
↳ travel, and Knock Block frame protection. Equipped with Bontrager components and a
↳ Shimano XT drivetrain, this bike excels on technical trails and enduro race courses."
    }
]

bike_data = [{**d, "bike_embedding": emb_model.embed(d["description"])} for d in bike_
↳ data]

bike_schema = {
    "index": {
        "name": "bike-json",
        "prefix": "bike-json",
        "storage_type": "json", # JSON storage type
    }
}
```

(continues on next page)

(continued from previous page)

```

    },
    "fields": [
        {
            "name": "model",
            "type": "tag",
            "path": "$.metadata.model" # note the '$'
        },
        {
            "name": "brand",
            "type": "tag",
            "path": "$.metadata.brand"
        },
        {
            "name": "price",
            "type": "numeric",
            "path": "$.metadata.price"
        },
        {
            "name": "bike_embedding",
            "type": "vector",
            "attrs": {
                "dims": len(bike_data[0]["bike_embedding"]),
                "distance_metric": "cosine",
                "algorithm": "flat",
                "datatype": "float32"
            }
        }
    ]
}

```

```

# construct a search index from the json schema
bike_index = SearchIndex.from_dict(bike_schema, redis_url="redis://localhost:6379")

# create the index (no data yet)
bike_index.create(overwrite=True)

```

```
bike_index.load(bike_data)
```

```

['bike-json:01KHKJ5WW3DJE0X6E85GG27V0X',
 'bike-json:01KHKJ5WW3DJE0X6E85GG27V0Y']

```

```

from redisvl.query import VectorQuery

vec = emb_model.embed("I'd like a bike for aggressive riding")

v = VectorQuery(
    vector=vec,
    vector_field_name="bike_embedding",
    return_fields=[
        "brand",

```

(continues on next page)

(continued from previous page)

```

        "name",
        "$.metadata.type"
    ]
)

results = bike_index.query(v)

```

Note: As shown in the example if you want to retrieve a field from json object that was not indexed you will also need to supply the full path as with `$.metadata.type`.

```
results
```

```

[{'id': 'bike-json:01KHKJ5WW3DJE0X6E85GG27V0Y',
  'vector_distance': '0.519988954067',
  'brand': 'Trek',
  '$.metadata.type': 'Enduro bikes'},
 {'id': 'bike-json:01KHKJ5WW3DJE0X6E85GG27V0X',
  'vector_distance': '0.65762424469',
  'brand': 'Specialized',
  '$.metadata.type': 'Enduro bikes'}]

```

Next Steps

Now that you understand Hash vs JSON storage, explore these related guides:

- [Getting Started](#) - Learn the basics of RedisVL indexes and queries
- [Query and Filter Data](#) - Apply filters to narrow down search results
- [Use Advanced Query Types](#) - Explore TextQuery, HybridQuery, and more

```

# Cleanup
bike_index.delete()

```

Rerank Search Results

This guide demonstrates how to use RedisVL to rerank search results (documents, chunks, or records) based on query relevance. RedisVL supports reranking through Cross-Encoders from [Hugging Face](#), [Cohere Rerank API](#), and [VoyageAI Rerank API](#).

Prerequisites

Before you begin, ensure you have:

- Installed RedisVL: `pip install redisvl`
- A running Redis instance ([Redis 8+](#) or [Redis Cloud](#))

What You'll Learn

By the end of this guide, you will be able to:

- Rerank search results using HuggingFace Cross-Encoders
- Use the Cohere Rerank API with search results
- Use the VoyageAI Rerank API for result reranking
- Control the number of returned results after reranking

```
# import necessary modules
import os
```

Simple Reranking

Reranking provides a relevance boost to search results generated by traditional (lexical) or semantic search strategies.

As a simple demonstration, take the passages and user query below:

```
query = "What is the capital of the United States?"
docs = [
    "Carson City is the capital city of the American state of Nevada. At the 2010 United_
↪States Census, Carson City had a population of 55,274.",
    "The Commonwealth of the Northern Mariana Islands is a group of islands in the_
↪Pacific Ocean that are a political division controlled by the United States. Its_
↪capital is Saipan.",
    "Charlotte Amalie is the capital and largest city of the United States Virgin_
↪Islands. It has about 20,000 people. The city is on the island of Saint Thomas.",
    "Washington, D.C. (also known as simply Washington or D.C., and officially as the_
↪District of Columbia) is the capital of the United States. It is a federal district._
↪The President of the USA and many major national government offices are in the_
↪territory. This makes it the political center of the United States of America.",
    "Capital punishment (the death penalty) has existed in the United States since_
↪before the United States was a country. As of 2017, capital punishment is legal in 30_
↪of the 50 states. The federal government (including the United States military) also_
↪uses capital punishment."
]
```

The goal of reranking is to provide a more fine-grained quality improvement to initial search results. With RedisVL, this would likely be results coming back from a search operation like full text or vector.

Using the Cross-Encoder Reranker

To use the cross-encoder reranker we initialize an instance of `HFCrossEncoderReranker` passing a suitable model (if no model is provided, the `cross-encoder/ms-marco-MiniLM-L-6-v2` model is used):

```
from redisvl.utils.rerank import HFCrossEncoderReranker

cross_encoder_reranker = HFCrossEncoderReranker("BAAI/bge-reranker-base")
```

Rerank documents with HFCrossEncoderReranker

With the obtained reranker instance we can rerank and truncate the list of documents based on relevance to the initial query.

```
results, scores = cross_encoder_reranker.rank(query=query, docs=docs)
```

```
for result, score in zip(results, scores):
    print(score, " -- ", result)
```

```
0.07461103051900864 -- {'content': 'Washington, D.C. (also known as simply Washington,
↳ or D.C., and officially as the District of Columbia) is the capital of the United
↳ States. It is a federal district. The President of the USA and many major national
↳ government offices are in the territory. This makes it the political center of the
↳ United States of America.'}
0.052202966064214706 -- {'content': 'Charlotte Amalie is the capital and largest city
↳ of the United States Virgin Islands. It has about 20,000 people. The city is on the
↳ island of Saint Thomas.'}
0.3802356719970703 -- {'content': 'Carson City is the capital city of the American
↳ state of Nevada. At the 2010 United States Census, Carson City had a population of 55,
↳ 274.'}
```

Using the Cohere Reranker

To initialize the Cohere reranker you'll need to install the cohere library and provide the right Cohere API Key.

```
#!/pip install cohere
```

```
import getpass

# setup the API Key
api_key = os.environ.get("COHERE_API_KEY") or getpass.getpass("Enter your Cohere API
↳ key: ")
```

```
from redisvl.utils.rerank import CohereReranker

cohere_reranker = CohereReranker(limit=3, api_config={"api_key": api_key})
```

Rerank documents with CohereReranker

The following example uses CohereReranker to rerank and truncate the list of documents based on relevance to the initial query.

```
results, scores = cohere_reranker.rank(query=query, docs=docs)
```

```
for result, score in zip(results, scores):
    print(score, " -- ", result)
```

```
0.9990564 -- Washington, D.C. (also known as simply Washington or D.C., and officially
↳as the District of Columbia) is the capital of the United States. It is a federal
↳district. The President of the USA and many major national government offices are in
↳the territory. This makes it the political center of the United States of America.
0.7516481 -- Capital punishment (the death penalty) has existed in the United States
↳since before the United States was a country. As of 2017, capital punishment is legal
↳in 30 of the 50 states. The federal government (including the United States military)
↳also uses capital punishment.
0.08882029 -- The Commonwealth of the Northern Mariana Islands is a group of islands
↳in the Pacific Ocean that are a political division controlled by the United States.
↳Its capital is Saipan.
```

Working with semi-structured documents

Often times the initial result set includes other metadata and components that could be used to steer the reranking relevancy. To accomplish this, we can set the `rank_by` argument and provide documents with those additional fields.

```
docs = [
    {
        "source": "wiki",
        "passage": "Carson City is the capital city of the American state of Nevada. At
↳the 2010 United States Census, Carson City had a population of 55,274."
    },
    {
        "source": "encyclopedia",
        "passage": "The Commonwealth of the Northern Mariana Islands is a group of
↳islands in the Pacific Ocean that are a political division controlled by the United
↳States. Its capital is Saipan."
    },
    {
        "source": "textbook",
        "passage": "Charlotte Amalie is the capital and largest city of the United
↳States Virgin Islands. It has about 20,000 people. The city is on the island of Saint
↳Thomas."
    },
    {
        "source": "textbook",
        "passage": "Washington, D.C. (also known as simply Washington or D.C., and
↳officially as the District of Columbia) is the capital of the United States. It is a
↳federal district. The President of the USA and many major national government offices
↳are in the territory. This makes it the political center of the United States of
```

(continues on next page)

(continued from previous page)

```

    ↪America."
    },
    {
        "source": "wiki",
        "passage": "Capital punishment (the death penalty) has existed in the United
    ↪States since before the United States was a country. As of 2017, capital punishment is
    ↪legal in 30 of the 50 states. The federal government (including the United States
    ↪military) also uses capital punishment."
    }
]

```

```

results, scores = cohere_reranker.rank(query=query, docs=docs, rank_by=["passage",
    ↪"source"])

```

```

for result, score in zip(results, scores):
    print(score, " -- ", result)

```

```

0.9988121 -- {'source': 'textbook', 'passage': 'Washington, D.C. (also known as simply
    ↪Washington or D.C., and officially as the District of Columbia) is the capital of the
    ↪United States. It is a federal district. The President of the USA and many major
    ↪national government offices are in the territory. This makes it the political center
    ↪of the United States of America.'}
0.5974905 -- {'source': 'wiki', 'passage': 'Capital punishment (the death penalty) has
    ↪existed in the United States since before the United States was a country. As of 2017,
    ↪capital punishment is legal in 30 of the 50 states. The federal government (including
    ↪the United States military) also uses capital punishment.'}
0.059101548 -- {'source': 'encyclopedia', 'passage': 'The Commonwealth of the Northern
    ↪Mariana Islands is a group of islands in the Pacific Ocean that are a political
    ↪division controlled by the United States. Its capital is Saipan.'}

```

Using the VoyageAI Reranker

To initialize the VoyageAI reranker you'll need to install the voyaeai library and provide the right VoyageAI API Key.

```

#!pip install voyageai

```

```

import getpass

# setup the API Key
api_key = os.environ.get("VOYAGE_API_KEY") or getpass.getpass("Enter your VoyageAI API
    ↪key: ")

```

```

from redisvl.utils.rerank import VoyageAIReranker

reranker = VoyageAIReranker(model="rerank-2.5", limit=3, api_config={"api_key": api_key})
# Please check the available models at https://docs.voyageai.com/docs/reranker

```

Rerank documents with VoyageAIReranker

The following example uses VoyageAIReranker to rerank and truncate the list of documents based on relevance to the initial query.

```
results, scores = reranker.rank(query=query, docs=docs)
```

```
for result, score in zip(results, scores):  
    print(score, " -- ", result)
```

```
0.796875 -- Washington, D.C. (also known as simply Washington or D.C., and officially  
↳as the District of Columbia) is the capital of the United States. It is a federal  
↳district. The President of the USA and many major national government offices are in  
↳the territory. This makes it the political center of the United States of America.  
0.578125 -- Charlotte Amalie is the capital and largest city of the United States  
↳Virgin Islands. It has about 20,000 people. The city is on the island of Saint Thomas.  
0.5625 -- Carson City is the capital city of the American state of Nevada. At the 2010  
↳United States Census, Carson City had a population of 55,274.
```

Next Steps

Now that you understand reranking, explore these related guides:

- *Create Embeddings with Vectorizers* - Generate embeddings using various providers
- *Query and Filter Data* - Build complex filter expressions for search
- *Use Advanced Query Types* - Learn about HybridQuery and other query types

Cleanup

This guide does not create a persistent index, so no cleanup is required.

Optimize Indexes with SVS-VAMANA

This guide covers SVS-VAMANA (Scalable Vector Search with VAMANA graph algorithm), a graph-based vector search algorithm optimized for compression methods to reduce memory usage. It combines the Vamana graph algorithm with advanced compression techniques (LVQ and LeanVec) and is optimized for Intel hardware.

Prerequisites

Before you begin, ensure you have:

- Installed RedisVL: `pip install redisvl`
- A running Redis instance with Redis `>= 8.2.0` and Redis Search `>= 2.8.10` ([Redis 8+](#) or [Redis Cloud](#))

Note: SVS-VAMANA only supports FLOAT16 and FLOAT32 datatypes.

What You'll Learn

By the end of this guide, you will be able to:

- Understand when to use SVS-VAMANA for vector search
- Configure compression settings for memory optimization
- Use the CompressionAdvisor for automatic optimization
- Trade off between memory usage, speed, and search quality

SVS-VAMANA offers:

- **Fast approximate nearest neighbor search** using graph-based algorithms
- **Vector compression** (LVQ, LeanVec) with up to 87.5% memory savings
- **Dimensionality reduction** (optional, with LeanVec)
- **Automatic performance optimization** through CompressionAdvisor

Use SVS-VAMANA when:

- Large datasets where memory is expensive
- Cloud deployments with memory-based pricing
- When 90-95% recall is acceptable
- High-dimensional vectors (>1024 dims) with LeanVec compression

```
# Import necessary modules
import numpy as np
from redisvl.index import SearchIndex
from redisvl.query import VectorQuery
from redisvl.utils import CompressionAdvisor
from redisvl.redis.utils import array_to_buffer

# Set random seed for reproducible results
np.random.seed(42)
```

```
# Redis connection
REDIS_URL = "redis://localhost:6379"
```

Quick Start with CompressionAdvisor

The easiest way to get started with SVS-VAMANA is using the CompressionAdvisor utility, which automatically recommends optimal configuration based on your vector dimensions and performance priorities.

```
# Get recommended configuration for common embedding dimensions
dims = 1024 # Common embedding dimensions (works reliably with SVS-VAMANA)

config = CompressionAdvisor.recommend(
    dims=dims,
    priority="balanced" # Options: "memory", "speed", "balanced"
)
```

(continues on next page)

(continued from previous page)

```

print("Recommended Configuration:")
for key, value in config.model_dump().items():
    print(f"  {key}: {value}")

# Estimate memory savings
savings = CompressionAdvisor.estimate_memory_savings(
    config.compression,
    dims,
    config.reduce
)
print(f"\nEstimated Memory Savings: {savings}%")

```

```

Recommended Configuration:
  algorithm: svsv-vamana
  datatype: float16
  compression: LeanVec4x8
  reduce: 512
  graph_max_degree: 64
  construction_window_size: 300
  search_window_size: 30

Estimated Memory Savings: 81.2%

```

Creating an SVS-VAMANA Index

Let's create an index using the recommended configuration. We'll use a simple schema with text content and vector embeddings.

```

# Create index schema with recommended SVS-VAMANA configuration
config_dict = config.model_dump(exclude_none=True)
schema = {
    "index": {
        "name": "svs_demo",
        "prefix": "doc",
    },
    "fields": [
        {"name": "content", "type": "text"},
        {"name": "category", "type": "tag"},
        {
            "name": "embedding",
            "type": "vector",
            "attrs": {
                "dims": dims,
                **config_dict, # Use the recommended configuration
                "distance_metric": "cosine"
            }
        }
    ]
}

```

(continues on next page)

(continued from previous page)

```
# Create the index
index = SearchIndex.from_dict(schema, redis_url=REDIS_URL)
index.create(overwrite=True)

print(f" Created SVS-VAMANA index: {index.name}")
print(f"   Algorithm: {config.algorithm}")
print(f"   Compression: {config.compression}")
print(f"   Dimensions: {dims}")
if config.reduce is not None:
    print(f"   Reduced to: {config.reduce} dimensions")
```

```
Created SVS-VAMANA index: svd_demo
Algorithm: svd-vamana
Compression: LeanVec4x8
Dimensions: 1024
Reduced to: 512 dimensions
```

Loading Sample Data

Let's create some sample documents with embeddings to demonstrate SVS-VAMANA search capabilities.

```
# Generate sample data
sample_documents = [
    {"content": "Machine learning algorithms for data analysis", "category": "technology"},
    {"content": "Natural language processing and text understanding", "category": "technology"},
    {"content": "Computer vision and image recognition systems", "category": "technology"},
    {"content": "Delicious pasta recipes from Italy", "category": "food"},
    {"content": "Traditional French cooking techniques", "category": "food"},
    {"content": "Healthy meal planning and nutrition", "category": "food"},
    {"content": "Travel guide to European destinations", "category": "travel"},
    {"content": "Adventure hiking in mountain regions", "category": "travel"},
    {"content": "Cultural experiences in Asian cities", "category": "travel"},
    {"content": "Financial planning for retirement", "category": "finance"},
]

# Generate random embeddings for demonstration
# In practice, you would use a real embedding model
data_to_load = []

# Use reduced dimensions if LeanVec compression is applied
vector_dims = config.reduce if config.reduce is not None else dims
print(f"Creating vectors with {vector_dims} dimensions (reduced from {dims} if applicable)")

for i, doc in enumerate(sample_documents):
    # Create a random vector with some category-based clustering
    base_vector = np.random.random(vector_dims).astype(np.float32)
```

(continues on next page)

(continued from previous page)

```

# Add some category-based similarity (optional, for demo purposes)
category_offset = hash(doc["category"]) % 100 / 1000.0
base_vector[0] += category_offset

# Convert to the datatype specified in config
if config.datatype == "float16":
    base_vector = base_vector.astype(np.float16)

data_to_load.append({
    "content": doc["content"],
    "category": doc["category"],
    "embedding": array_to_buffer(base_vector, dtype=config.datatype)
})

# Load data into the index
index.load(data_to_load)
print(f" Loaded {len(data_to_load)} documents into the index")

# Wait a moment for indexing to complete
import time
time.sleep(2)

# Verify the data was loaded
info = index.info()
print(f"   Index now contains {info.get('num_docs', 0)} documents")

```

```

Creating vectors with 512 dimensions (reduced from 1024 if applicable)
Loaded 10 documents into the index
Index now contains 0 documents

```

Performing Vector Searches

Now let's perform some vector similarity searches using our SVS-VAMANA index.

```

# Create a query vector (in practice, this would be an embedding of your query text)
# Important: Query vector must match the index datatype and dimensions
vector_dims = config.reduce if config.reduce is not None else dims
if config.datatype == "float16":
    query_vector = np.random.random(vector_dims).astype(np.float16)
else:
    query_vector = np.random.random(vector_dims).astype(np.float32)

# Perform a vector similarity search
query = VectorQuery(
    vector=query_vector.tolist(),
    vector_field_name="embedding",
    return_fields=["content", "category"],
    num_results=5
)

```

(continues on next page)

(continued from previous page)

```

results = index.query(query)

print(" Vector Search Results:")
print("=" * 50)
for i, result in enumerate(results, 1):
    distance = result.get('vector_distance', 'N/A')
    print(f"{i}. [{result['category']}] {result['content']}")
    print(f"    Distance: {distance:.4f}" if isinstance(distance, (int, float)) else f"    ↪Distance: {distance}")
    print()

```

Vector Search Results:

=====

Runtime Parameters for Performance Tuning

SVS-VAMANA supports runtime parameters that can be adjusted at query time without rebuilding the index. These parameters allow you to fine-tune the trade-off between search speed and accuracy.

Available Runtime Parameters:

- **search_window_size:** Controls the size of the search window during KNN search (higher = better recall, slower search)
- **epsilon:** Approximation factor for range queries (default: 0.01)
- **use_search_history:** Whether to use search buffer (OFF/ON/AUTO, default: AUTO)
- **search_buffer_capacity:** Tuning parameter for 2-level compression (default: search_window_size)

Let's see how these parameters affect search performance:

```

# Example 1: Basic query with default parameters
basic_query = VectorQuery(
    vector=query_vector.tolist(),
    vector_field_name="embedding",
    return_fields=["content", "category"],
    num_results=5
)

print(" Basic Query (default parameters):")
results = index.query(basic_query)
print(f"Found {len(results)} results\n")

# Example 2: Query with tuned runtime parameters for higher recall
tuned_query = VectorQuery(
    vector=query_vector.tolist(),
    vector_field_name="embedding",
    return_fields=["content", "category"],
    num_results=5,
    search_window_size=40,      # Larger window for better recall
    use_search_history='ON',   # Use search history
)

```

(continues on next page)

(continued from previous page)

```

    search_buffer_capacity=50    # Larger buffer capacity
)

print(" Tuned Query (higher recall parameters):")
results = index.query(tuned_query)
print(f"Found {len(results)} results")
print("\nNote: Higher search_window_size improves recall but may increase latency")

```

```

Basic Query (default parameters):
Found 0 results

```

```

Tuned Query (higher recall parameters):
Found 0 results

```

```

Note: Higher search_window_size improves recall but may increase latency

```

Range Queries with SVS-VAMANA

Range queries find all vectors within a certain distance threshold. For range queries, you can use the `epsilon` parameter to control the approximation factor:

```

from redisvl.query import VectorRangeQuery

# Range query with epsilon parameter for approximation control
# Note: search_window_size and use_search_history are only supported for KNN queries.
# (VectorQuery),
# not for range queries (VectorRangeQuery). Use epsilon to control the approximation
# factor.
range_query = VectorRangeQuery(
    vector=query_vector.tolist(),
    vector_field_name="embedding",
    return_fields=["content", "category"],
    distance_threshold=0.3,
    epsilon=0.05,           # Approximation factor for range queries
)

results = index.query(range_query)
print(f" Range Query Results: Found {len(results)} vectors within distance threshold 0.3")
for i, result in enumerate(results[:3], 1):
    distance = result.get('vector_distance', 'N/A')
    print(f"{i}. {result['content'][:50]}... (distance: {distance})")

```

```

Range Query Results: Found 0 vectors within distance threshold 0.3

```

Understanding Compression Types

SVS-VAMANA supports different compression algorithms that trade off between memory usage and search quality. Let's explore the available options.

```
# Compare different compression priorities
print("Compression Recommendations for Different Priorities:")
print("=" * 60)

priorities = ["memory", "speed", "balanced"]
for priority in priorities:
    config = CompressionAdvisor.recommend(dims=dims, priority=priority)
    savings = CompressionAdvisor.estimate_memory_savings(
        config.compression,
        dims,
        config.reduce
    )

    print(f"\n{priority.upper()} Priority:")
    print(f"  Compression: {config.compression}")
    print(f"  Datatype: {config.datatype}")
    if config.reduce is not None:
        print(f"  Dimensionality reduction: {dims} → {config.reduce}")
    print(f"  Search window size: {config.search_window_size}")
    print(f"  Memory savings: {savings}%")
```

```
Compression Recommendations for Different Priorities:
```

```
=====

MEMORY Priority:
  Compression: LeanVec4x8
  Datatype: float16
  Dimensionality reduction: 1024 → 512
  Search window size: 20
  Memory savings: 81.2%

SPEED Priority:
  Compression: LeanVec4x8
  Datatype: float16
  Dimensionality reduction: 1024 → 256
  Search window size: 40
  Memory savings: 90.6%

BALANCED Priority:
  Compression: LeanVec4x8
  Datatype: float16
  Dimensionality reduction: 1024 → 512
  Search window size: 30
  Memory savings: 81.2%
```

Compression Types Explained

SVS-VAMANA offers several compression algorithms:

LVQ (Learned Vector Quantization)

- **LVQ4**: 4 bits per dimension (87.5% memory savings)
- **LVQ4x4**: 8 bits per dimension (75% memory savings)
- **LVQ4x8**: 12 bits per dimension (62.5% memory savings)
- **LVQ8**: 8 bits per dimension (75% memory savings)

LeanVec (Compression + Dimensionality Reduction)

- **LeanVec4x8**: 12 bits per dimension + dimensionality reduction
- **LeanVec8x8**: 16 bits per dimension + dimensionality reduction

The CompressionAdvisor automatically chooses the best compression type based on your vector dimensions and priority.

```
# Demonstrate compression savings for different vector dimensions
test_dimensions = [384, 768, 1024, 1536, 3072]

print("Memory Savings by Vector Dimension:")
print("=" * 50)
print(f"{'Dims':<6} {'Compression':<12} {'Savings':<8} {'Strategy'}")
print("-" * 50)

for dims in test_dimensions:
    config = CompressionAdvisor.recommend(dims=dims, priority="balanced")
    savings = CompressionAdvisor.estimate_memory_savings(
        config.compression,
        dims,
        config.reduce
    )

    strategy = "LeanVec" if dims >= 1024 else "LVQ"
    print(f"{'dims':<6} {'config.compression':<12} {'savings:>6.1f}% {'strategy'}")
```

Memory Savings by Vector Dimension:

```
=====
Dims    Compression  Savings  Strategy
-----
384     LVQ4x4         75.0%   LVQ
768     LVQ4x4         75.0%   LVQ
1024    LeanVec4x8     81.2%   LeanVec
1536    LeanVec4x8     81.2%   LeanVec
3072    LeanVec4x8     81.2%   LeanVec
```

Hybrid Queries with SVS-VAMANA

SVS-VAMANA can be combined with other Redis search capabilities for powerful hybrid queries that filter by metadata while performing vector similarity search.

```
# Perform a hybrid search: vector similarity + category filter
hybrid_query = VectorQuery(
    vector=query_vector.tolist(),
    vector_field_name="embedding",
    return_fields=["content", "category"],
    num_results=3
)

# Add a filter to only search within "technology" category
hybrid_query.set_filter("@category:{technology}")

filtered_results = index.query(hybrid_query)

print(" Hybrid Search Results (Technology category only):")
print("=" * 55)
for i, result in enumerate(filtered_results, 1):
    distance = result.get('vector_distance', 'N/A')
    print(f"{i}. [{result['category']}] {result['content']}")
    print(f"    Distance: {distance:.4f}" if isinstance(distance, (int, float)) else f"    ↪Distance: {distance}")
    print()
```

```
Hybrid Search Results (Technology category only):
=====
```

Performance Monitoring

Let's examine the index statistics to understand the performance characteristics of our SVS-VAMANA index.

```
# Get detailed index information
info = index.info()

print(" Index Statistics:")
print("=" * 30)
print(f"Documents: {info.get('num_docs', 0)}")

# Handle vector_index_sz_mb which might be a string
vector_size = info.get('vector_index_sz_mb', 0)
if isinstance(vector_size, str):
    try:
        vector_size = float(vector_size)
    except ValueError:
        vector_size = 0.0
print(f"Vector index size: {vector_size:.2f} MB")

# Handle total_indexing_time which might also be a string
indexing_time = info.get('total_indexing_time', 0)
```

(continues on next page)

(continued from previous page)

```

if isinstance(indexing_time, str):
    try:
        indexing_time = float(indexing_time)
    except ValueError:
        indexing_time = 0.0
print(f"Total indexing time: {indexing_time:.2f} seconds")

# Calculate memory efficiency
if info.get('num_docs', 0) > 0 and vector_size > 0:
    mb_per_doc = vector_size / info.get('num_docs', 1)
    print(f"Memory per document: {mb_per_doc:.4f} MB")

    # Estimate for larger datasets
    for scale in [1000, 10000, 100000]:
        estimated_mb = mb_per_doc * scale
        print(f"Estimated size for {scale:,} docs: {estimated_mb:.1f} MB")
else:
    print("Memory efficiency calculation requires documents and vector index size > 0")

```

Index Statistics:

```

=====
Documents: 0
Vector index size: 0.00 MB
Total indexing time: 0.27 seconds
Memory efficiency calculation requires documents and vector index size > 0

```

Advanced Manual Configuration

For advanced users who want full control over SVS-VAMANA parameters, you can manually configure the algorithm instead of using CompressionAdvisor.

```

# Example of manual SVS-VAMANA configuration
manual_schema = {
    "index": {
        "name": "svs_manual",
        "prefix": "manual",
    },
    "fields": [
        {"name": "content", "type": "text"},
        {
            "name": "embedding",
            "type": "vector",
            "attrs": {
                "dims": 768,
                "algorithm": "svs-vamana",
                "datatype": "float32",
                "distance_metric": "cosine",

                # Graph construction parameters
                "graph_max_degree": 64,
            }
        }
    ]
}

```

(continues on next page)

(continued from previous page)

```

        "construction_window_size": 300, # Higher = better quality, slower build

        # Search parameters
        "search_window_size": 40, # Higher = better recall, slower search

        # Compression settings
        "compression": "LVQ4x4", # Choose compression type
        "training_threshold": 10000, # Min vectors before compression
    }
}

↪ training

print("Manual SVS-VAMANA Configuration:")
print("=" * 40)
vector_attrs = manual_schema["fields"][1]["attrs"]
for key, value in vector_attrs.items():
    if key != "dims": # Skip dims as it's obvious
        print(f" {key}: {value}")

# Calculate memory savings for this configuration
manual_savings = CompressionAdvisor.estimate_memory_savings(
    "LVQ4x4", 768, None
)
print(f"\nEstimated memory savings: {manual_savings}%")

```

Manual SVS-VAMANA Configuration:

```

=====
algorithm: svsvamana
datatype: float32
distance_metric: cosine
graph_max_degree: 64
construction_window_size: 300
search_window_size: 40
compression: LVQ4x4
training_threshold: 10000

```

Estimated memory savings: 75.0%

Best Practices and Tips

When to Use SVS-VAMANA

- **Large datasets** (>10K vectors) where memory efficiency matters
- **High-dimensional vectors** (>512 dimensions) that benefit from compression
- **Applications** that can tolerate slight recall trade-offs for speed and memory savings

Parameter Tuning Guidelines

Index-time parameters (set during index creation):

- **Start with CompressionAdvisor** recommendations for compression and datatype
- Use **LeanVec** for high-dimensional vectors (1024 dims)
- Use **LVQ** for lower-dimensional vectors (<1024 dims)
- **graph_max_degree**: Higher values improve recall but increase memory usage
- **construction_window_size**: Higher values improve index quality but slow down build time

Runtime parameters (adjustable at query time without rebuilding index):

- **search_window_size**: Start with 20, increase to 40-100 for higher recall
- **epsilon**: Use 0.01-0.05 for range queries (higher = faster but less accurate)
- **use_search_history**: Use 'AUTO' (default) or 'ON' for better recall
- **search_buffer_capacity**: Usually set equal to search_window_size

Performance Considerations

- **Index build time** increases with higher construction_window_size
- **Search latency** increases with higher search_window_size (tunable at query time!)
- **Memory usage** decreases with more aggressive compression
- **Recall quality** may decrease with more aggressive compression or lower search_window_size

Next Steps

Now that you understand SVS-VAMANA optimization, explore these related guides:

- *Getting Started* - Learn the basics of RedisVL indexes
- *Choose a Storage Type* - Understand Hash vs JSON storage
- *Query and Filter Data* - Apply filters to narrow down search results

Cleanup

Clean up the indices created in this demo.

```
# Clean up demo indices
try:
    index.delete()
    print("Cleaned up svd_demo index")
except:
    print("- svd_demo index was already deleted or doesn't exist")
```

Cache Embeddings

RedisVL provides an `EmbeddingsCache` that stores and retrieves embedding vectors with their associated text and metadata. This cache is useful for applications that frequently compute the same embeddings.

Prerequisites

Before you begin, ensure you have:

- Installed RedisVL: `pip install redisvl`
- A running Redis instance ([Redis 8+](#) or [Redis Cloud](#))

What You'll Learn

By the end of this guide, you will be able to:

- Store and retrieve embedding vectors with the `EmbeddingsCache`
- Use batch operations for efficient processing
- Configure time-to-live (TTL) for cache entries
- Integrate the cache with vectorizers for automatic caching

Setup

First, let's import the necessary libraries. We'll use a text embedding model from HuggingFace to generate our embeddings.

```
import os
import time
import numpy as np

# Disable tokenizers parallelism to avoid deadlocks
os.environ["TOKENIZERS_PARALLELISM"] = "False"

# Import the EmbeddingsCache
from redisvl.extensions.cache.embeddings import EmbeddingsCache
from redisvl.utils.vectorize import HFTextVectorizer
```

Let's create a vectorizer to generate embeddings for our texts:

```
# Initialize the vectorizer
vectorizer = HFTextVectorizer(
    model="redis/langcache-embed-v1",
    cache_folder=os.getenv("SENTENCE_TRANSFORMERS_HOME")
)
```

Initializing the EmbeddingsCache

Now let's initialize our EmbeddingsCache. The cache requires a Redis connection to store the embeddings and their associated data.

```
# Initialize the embeddings cache
cache = EmbeddingsCache(
    name="embedcache",           # name prefix for Redis keys
    redis_url="redis://localhost:6379", # Redis connection URL
    ttl=None                     # Optional TTL in seconds (None means no
    ↪ expiration)
)
```

Basic Usage

Storing Embeddings

Let's store some text with its embedding in the cache. The set method takes the following parameters:

- **content**: The input text that was embedded
- **model_name**: The name of the embedding model used
- **embedding**: The embedding vector
- **metadata**: Optional metadata associated with the embedding
- **ttl**: Optional time-to-live override for this specific entry

```
# Text to embed
text = "What is machine learning?"
model_name = "redis/langcache-embed-v1"

# Generate the embedding
embedding = vectorizer.embed(text)

# Optional metadata
metadata = {"category": "ai", "source": "user_query"}

# Store in cache
key = cache.set(
    content=text,
    model_name=model_name,
    embedding=embedding,
    metadata=metadata
)

print(f"Stored with key: {key[:15]}...")
```

```
Stored with key: embedcache:909f...
```

Retrieving Embeddings

To retrieve an embedding from the cache, use the `get` method with the original text and model name:

```
# Retrieve from cache

if result := cache.get(content=text, model_name=model_name):
    print(f"Found in cache: {result['content']}")
    print(f"Model: {result['model_name']}")
    print(f"Metadata: {result['metadata']}")
    print(f"Embedding shape: {np.array(result['embedding']).shape}")
else:
    print("Not found in cache.")
```

```
Found in cache: What is machine learning?
Model: redis/langcache-embed-v1
Metadata: {'category': 'ai', 'source': 'user_query'}
Embedding shape: (768,)
```

Checking Existence

You can check if an embedding exists in the cache without retrieving it using the `exists` method:

```
# Check if existing text is in cache
exists = cache.exists(content=text, model_name=model_name)
print(f"First query exists in cache: {exists}")

# Check if a new text is in cache
new_text = "What is deep learning?"
exists = cache.exists(content=new_text, model_name=model_name)
print(f"New query exists in cache: {exists}")
```

```
First query exists in cache: True
New query exists in cache: False
```

Removing Entries

To remove an entry from the cache, use the `drop` method:

```
# Remove from cache
cache.drop(content=text, model_name=model_name)

# Verify it's gone
exists = cache.exists(content=text, model_name=model_name)
print(f"After dropping: {exists}")
```

```
After dropping: False
```

Advanced Usage

Key-Based Operations

The EmbeddingsCache also provides methods that work directly with Redis keys, which can be useful for advanced use cases:

```
# Store an entry again
key = cache.set(
    content=text,
    model_name=model_name,
    embedding=embedding,
    metadata=metadata
)
print(f"Stored with key: {key[:15]}...")

# Check existence by key
exists_by_key = cache.exists_by_key(key)
print(f"Exists by key: {exists_by_key}")

# Retrieve by key
result_by_key = cache.get_by_key(key)
print(f"Retrieved by key: {result_by_key['content']}")

# Drop by key
cache.drop_by_key(key)
```

```
Stored with key: embedcache:909f...
Exists by key: True
Retrieved by key: What is machine learning?
```

Batch Operations

When working with multiple embeddings, batch operations can significantly improve performance by reducing network roundtrips. The EmbeddingsCache provides methods prefixed with `m` (for “multi”) that handle batches efficiently.

```
# Create multiple embeddings
texts = [
    "What is machine learning?",
    "How do neural networks work?",
    "What is deep learning?"
]
embeddings = [vectorizer.embed(t) for t in texts]

# Prepare batch items as dictionaries
batch_items = [
    {
        "content": texts[0],
        "model_name": model_name,
        "embedding": embeddings[0],
        "metadata": {"category": "ai", "type": "question"}
```

(continues on next page)

(continued from previous page)

```

    },
    {
        "content": texts[1],
        "model_name": model_name,
        "embedding": embeddings[1],
        "metadata": {"category": "ai", "type": "question"}
    },
    {
        "content": texts[2],
        "model_name": model_name,
        "embedding": embeddings[2],
        "metadata": {"category": "ai", "type": "question"}
    }
]

# Store multiple embeddings in one operation
keys = cache.mset(batch_items)
print(f"Stored {len(keys)} embeddings with batch operation")

# Check if multiple embeddings exist in one operation
exist_results = cache.mexists(texts, model_name)
print(f"All embeddings exist: {all(exist_results)}")

# Retrieve multiple embeddings in one operation
results = cache.mget(texts, model_name)
print(f"Retrieved {len(results)} embeddings in one operation")

# Delete multiple embeddings in one operation
cache.mdrop(texts, model_name)

# Alternative: key-based batch operations
# cache.mget_by_keys(keys)      # Retrieve by keys
# cache.mexists_by_keys(keys)   # Check existence by keys
# cache.mdrop_by_keys(keys)     # Delete by keys

```

```

Stored 3 embeddings with batch operation
All embeddings exist: True
Retrieved 3 embeddings in one operation

```

Batch operations are particularly beneficial when working with large numbers of embeddings. They provide the same functionality as individual operations but with better performance by reducing network roundtrips.

For asynchronous applications, async versions of all batch methods are also available with the `am` prefix (e.g., `amset`, `amget`, `amexists`, `amdrop`).

Working with TTL (Time-To-Live)

You can set a global TTL when initializing the cache, or specify TTL for individual entries:

```
# Create a cache with a default 5-second TTL
ttl_cache = EmbeddingsCache(
    name="ttl_cache",
    redis_url="redis://localhost:6379",
    ttl=5 # 5 second TTL
)

# Store an entry
key = ttl_cache.set(
    content=text,
    model_name=model_name,
    embedding=embedding
)

# Check if it exists
exists = ttl_cache.exists_by_key(key)
print(f"Immediately after setting: {exists}")

# Wait for it to expire
time.sleep(6)

# Check again
exists = ttl_cache.exists_by_key(key)
print(f"After waiting: {exists}")
```

```
Immediately after setting: True
After waiting: False
```

You can also override the default TTL for individual entries:

```
# Store an entry with a custom 1-second TTL
key1 = ttl_cache.set(
    content="Short-lived entry",
    model_name=model_name,
    embedding=embedding,
    ttl=1 # Override with 1 second TTL
)

# Store another entry with the default TTL (5 seconds)
key2 = ttl_cache.set(
    content="Default TTL entry",
    model_name=model_name,
    embedding=embedding
    # No TTL specified = uses the default 5 seconds
)

# Wait for 2 seconds
time.sleep(2)
```

(continues on next page)

(continued from previous page)

```
# Check both entries
exists1 = ttl_cache.exists_by_key(key1)
exists2 = ttl_cache.exists_by_key(key2)

print(f"Entry with custom TTL after 2 seconds: {exists1}")
print(f"Entry with default TTL after 2 seconds: {exists2}")

# Cleanup
ttl_cache.drop_by_key(key2)
```

```
Entry with custom TTL after 2 seconds: False
Entry with default TTL after 2 seconds: True
```

Async Support

The `EmbeddingsCache` provides async versions of all methods for use in async applications. The async methods are prefixed with a (e.g., `aset`, `aget`, `aexists`, `adrop`).

```
async def async_cache_demo():
    # Store an entry asynchronously
    key = await cache.aset(
        content="Async embedding",
        model_name=model_name,
        embedding=embedding,
        metadata={"async": True}
    )

    # Check if it exists
    exists = await cache.aexists_by_key(key)
    print(f"Async set successful? {exists}")

    # Retrieve it
    result = await cache.aget_by_key(key)
    success = result is not None and result["content"] == "Async embedding"
    print(f"Async get successful? {success}")

    # Remove it
    await cache.adrop_by_key(key)

# Run the async demo
await async_cache_demo()
```

```
Async set successful? True
Async get successful? True
```

Real-World Example

Let's build a simple embeddings caching system for a text classification task. We'll check the cache before computing new embeddings to save computation time.

```
# Create a fresh cache for this example
example_cache = EmbeddingsCache(
    name="example_cache",
    redis_url="redis://localhost:6379",
    ttl=3600 # 1 hour TTL
)

vectorizer = HFTextVectorizer(
    model=model_name,
    cache=example_cache,
    cache_folder=os.getenv("SENTENCE_TRANSFORMERS_HOME")
)

# Simulate processing a stream of queries
queries = [
    "What is artificial intelligence?",
    "How does machine learning work?",
    "What is artificial intelligence?", # Repeated query
    "What are neural networks?",
    "How does machine learning work?" # Repeated query
]

# Process the queries and track statistics
total_queries = 0
cache_hits = 0

for query in queries:
    total_queries += 1

    # Check cache before computing
    before = example_cache.exists(content=query, model_name=model_name)
    if before:
        cache_hits += 1

    # Get embedding (will compute or use cache)
    embedding = vectorizer.embed(query)

# Report statistics
cache_misses = total_queries - cache_hits
hit_rate = (cache_hits / total_queries) * 100

print("\nStatistics:")
print(f"Total queries: {total_queries}")
print(f"Cache hits: {cache_hits}")
print(f"Cache misses: {cache_misses}")
print(f"Cache hit rate: {hit_rate:.1f}%")

# Cleanup
```

(continues on next page)

(continued from previous page)

```
for query in set(queries): # Use set to get unique queries
    example_cache.drop(content=query, model_name=model_name)
```

You try to use a model that was created with version 4.1.0, however, your version is 3.4.
 ↳ 1. This might cause unexpected behavior or errors. In that case, try to update to the
 ↳ latest version.

```
Statistics:
Total queries: 5
Cache hits: 2
Cache misses: 3
Cache hit rate: 40.0%
```

Performance Benchmark

Let's run benchmarks to compare the performance of embedding with and without caching, as well as batch versus individual operations.

```
# Text to use for benchmarking
benchmark_text = "This is a benchmark text to measure the performance of embedding_
↳ caching."

# Create a fresh cache for benchmarking
benchmark_cache = EmbeddingsCache(
    name="benchmark_cache",
    redis_url="redis://localhost:6379",
    ttl=3600 # 1 hour TTL
)
vectorizer.cache = benchmark_cache

# Number of iterations for the benchmark
n_iterations = 10

# Benchmark without caching
print("Benchmarking without caching:")
start_time = time.time()
for _ in range(n_iterations):
    embedding = vectorizer.embed(text, skip_cache=True)
no_cache_time = time.time() - start_time
print(f"Time taken without caching: {no_cache_time:.4f} seconds")
print(f"Average time per embedding: {no_cache_time/n_iterations:.4f} seconds")

# Benchmark with caching
print("\nBenchmarking with caching:")
start_time = time.time()
for _ in range(n_iterations):
    embedding = vectorizer.embed(text)
cache_time = time.time() - start_time
print(f"Time taken with caching: {cache_time:.4f} seconds")
print(f"Average time per embedding: {cache_time/n_iterations:.4f} seconds")
```

(continues on next page)

(continued from previous page)

```
# Compare performance
speedup = no_cache_time / cache_time
latency_reduction = (no_cache_time/n_iterations) - (cache_time/n_iterations)
print(f"\nPerformance comparison:")
print(f"Speedup with caching: {speedup:.2f}x faster")
print(f"Time saved: {no_cache_time - cache_time:.4f} seconds ({(1 - cache_time/no_cache_
↪time) * 100:.1f}%)")
print(f"Latency reduction: {latency_reduction:.4f} seconds per query")
```

```
Benchmarking without caching:
Time taken without caching: 0.4564 seconds
Average time per embedding: 0.0456 seconds

Benchmarking with caching:
Time taken with caching: 0.0619 seconds
Average time per embedding: 0.0062 seconds

Performance comparison:
Speedup with caching: 7.37x faster
Time saved: 0.3945 seconds (86.4%)
Latency reduction: 0.0395 seconds per query
```

Common Use Cases for Embedding Caching

Embedding caching is particularly useful in the following scenarios:

1. **Search applications:** Cache embeddings for frequently searched queries to reduce latency
2. **Content recommendation systems:** Cache embeddings for content items to speed up similarity calculations
3. **API services:** Reduce costs and improve response times when generating embeddings through paid APIs
4. **Batch processing:** Speed up processing of datasets that contain duplicate texts
5. **Chatbots and virtual assistants:** Cache embeddings for common user queries to provide faster responses
6. **Development** workflows

Cleanup

Let's clean up our caches to avoid leaving data in Redis:

```
# Clean up all caches
cache.clear()
ttl_cache.clear()
example_cache.clear()
benchmark_cache.clear()
```

Next Steps

Now that you understand embeddings caching, explore these related guides:

- [Cache LLM Responses](#) - Cache full LLM responses based on semantic similarity
- [Create Embeddings with Vectorizers](#) - Learn about different embedding providers
- [Manage LLM Message History](#) - Store and retrieve conversation history

Summary

The `EmbeddingsCache` provides an efficient way to store and retrieve embeddings with their associated text and meta-data. Key features include:

- Simple API for storing and retrieving individual embeddings (`set/get`)
- Batch operations for working with multiple embeddings efficiently (`mset/mget/mexists/mdrop`)
- Support for metadata storage alongside embeddings
- Configurable time-to-live (TTL) for cache entries
- Key-based operations for advanced use cases
- Async support for use in asynchronous applications
- Significant performance improvements (15-20x faster with batch operations)

By using the `EmbeddingsCache`, you can reduce computational costs and improve the performance of applications that rely on embeddings.

Use Advanced Query Types

This guide covers advanced query types available in RedisVL:

1. **TextQuery**: Full text search with advanced scoring
2. **AggregateHybridQuery and HybridQuery**: Combines text and vector search for hybrid retrieval
3. **MultiVectorQuery**: Search over multiple vector fields simultaneously

These query types enable sophisticated search applications that go beyond simple vector similarity search.

Prerequisites

Before you begin, ensure you have:

- Installed RedisVL: `pip install redisvl`
- A running Redis instance ([Redis 8+](#) or [Redis Cloud](#))
- For `HybridQuery`: Redis `>= 8.4.0` and `redis-py >= 7.1.0`

What You'll Learn

By the end of this guide, you will be able to:

- Perform full-text search with `TextQuery` and advanced scoring options
- Combine text and vector search using `HybridQuery` and `AggregateHybridQuery`
- Search across multiple vector fields with `MultiVectorQuery`
- Configure custom stopwords for text search

Setup and Data Preparation

First, let's create a schema and prepare sample data that includes text fields, numeric fields, and vector fields.

```
import numpy as np
from jupyterutils import result_print

# Sample data with text descriptions, categories, and vectors
data = [
    {
        'product_id': 'prod_1',
        'brief_description': 'comfortable running shoes for athletes',
        'full_description': 'Engineered with a dual-layer EVA foam midsole and FlexWeave_
↳breathable mesh upper, these running shoes deliver responsive cushioning for long-
↳distance runs. The anatomical footbed adapts to your stride while the carbon rubber_
↳outsole provides superior traction on varied terrain.',
        'category': 'footwear',
        'price': 89.99,
        'rating': 4.5,
        'text_embedding': np.array([0.1, 0.2, 0.1], dtype=np.float32).tobytes(),
        'image_embedding': np.array([0.8, 0.1], dtype=np.float32).tobytes(),
    },
    {
        'product_id': 'prod_2',
        'brief_description': 'lightweight running jacket with water resistance',
        'full_description': 'Stay protected with this ultralight 2.5-layer DWR-coated_
↳shell featuring laser-cut ventilation zones and reflective piping for low-light_
↳visibility. Packs into its own chest pocket and weighs just 4.2 oz, making it ideal_
↳for unpredictable weather conditions.',
        'category': 'outerwear',
        'price': 129.99,
        'rating': 4.8,
        'text_embedding': np.array([0.2, 0.3, 0.2], dtype=np.float32).tobytes(),
        'image_embedding': np.array([0.7, 0.2], dtype=np.float32).tobytes(),
    },
    {
        'product_id': 'prod_3',
        'brief_description': 'professional tennis racket for competitive players',
        'full_description': 'Competition-grade racket featuring a 98 sq in head size,_
↳16x19 string pattern, and aerospace-grade graphite frame that delivers explosive power_
↳with pinpoint control. Tournament-approved specs include 315g weight and 68 RA_
↳stiffness rating for advanced baseline play.',
    },
]
```

(continues on next page)

(continued from previous page)

```

        'category': 'equipment',
        'price': 199.99,
        'rating': 4.9,
        'text_embedding': np.array([0.9, 0.1, 0.05], dtype=np.float32).tobytes(),
        'image_embedding': np.array([0.1, 0.9], dtype=np.float32).tobytes(),
    },
    {
        'product_id': 'prod_4',
        'brief_description': 'yoga mat with extra cushioning for comfort',
        'full_description': 'Premium 8mm thick TPE yoga mat with dual-texture surface -
↪smooth side for hot yoga flow and textured side for maximum grip during balancing
↪poses. Closed-cell technology prevents moisture absorption while alignment markers
↪guide proper positioning in asanas.',
        'category': 'accessories',
        'price': 39.99,
        'rating': 4.3,
        'text_embedding': np.array([0.15, 0.25, 0.15], dtype=np.float32).tobytes(),
        'image_embedding': np.array([0.5, 0.5], dtype=np.float32).tobytes(),
    },
    {
        'product_id': 'prod_5',
        'brief_description': 'basketball shoes with excellent ankle support',
        'full_description': 'High-top basketball sneakers with Zoom Air units in
↪forefoot and heel, reinforced lateral sidewalls for explosive cuts, and herringbone
↪traction pattern optimized for hardwood courts. The internal bootie construction and
↪extended ankle collar provide lockdown support during aggressive drives.',
        'category': 'footwear',
        'price': 139.99,
        'rating': 4.7,
        'text_embedding': np.array([0.12, 0.18, 0.12], dtype=np.float32).tobytes(),
        'image_embedding': np.array([0.75, 0.15], dtype=np.float32).tobytes(),
    },
    {
        'product_id': 'prod_6',
        'brief_description': 'swimming goggles with anti-fog coating',
        'full_description': 'Low-profile competition goggles with curved polycarbonate
↪lenses offering 180-degree peripheral vision and UV protection. Hydrophobic anti-fog
↪coating lasts 10x longer than standard treatments, while the split silicone strap and
↪interchangeable nose bridges ensure a watertight, custom fit.',
        'category': 'accessories',
        'price': 24.99,
        'rating': 4.4,
        'text_embedding': np.array([0.3, 0.1, 0.2], dtype=np.float32).tobytes(),
        'image_embedding': np.array([0.2, 0.8], dtype=np.float32).tobytes(),
    },
]

```

Define the Schema

Our schema includes:

- **Tag fields:** `product_id`, `category`
- **Text fields:** `brief_description` and `full_description` for full-text search
- **Numeric fields:** `price`, `rating`
- **Vector fields:** `text_embedding` (3 dimensions) and `image_embedding` (2 dimensions) for semantic search

```
schema = {
  "index": {
    "name": "advanced_queries",
    "prefix": "products",
    "storage_type": "hash",
  },
  "fields": [
    {"name": "product_id", "type": "tag"},
    {"name": "category", "type": "tag"},
    {"name": "brief_description", "type": "text"},
    {"name": "full_description", "type": "text"},
    {"name": "price", "type": "numeric"},
    {"name": "rating", "type": "numeric"},
    {
      "name": "text_embedding",
      "type": "vector",
      "attrs": {
        "dims": 3,
        "distance_metric": "cosine",
        "algorithm": "flat",
        "datatype": "float32"
      }
    },
    {
      "name": "image_embedding",
      "type": "vector",
      "attrs": {
        "dims": 2,
        "distance_metric": "cosine",
        "algorithm": "flat",
        "datatype": "float32"
      }
    }
  ],
}
```


Create Index and Load Data

```
from redisvl.index import SearchIndex

# Create the search index
index = SearchIndex.from_dict(schema, redis_url="redis://localhost:6379")

# Create the index and load data
index.create(overwrite=True)
keys = index.load(data)

print(f"Loaded {len(keys)} products into the index")
```

Loaded 6 products into the index

1. TextQuery: Full Text Search

The `TextQuery` class enables full text search with advanced scoring algorithms. It's ideal for keyword-based search with relevance ranking.

Basic Text Search

Let's search for products related to "running shoes":

```
from redisvl.query import TextQuery

# Create a text query
text_query = TextQuery(
    text="running shoes",
    text_field_name="brief_description",
    return_fields=["product_id", "brief_description", "category", "price"],
    num_results=5
)

results = index.query(text_query)
result_print(results)
```

<IPython.core.display.HTML object>

Text Search with Different Scoring Algorithms

RedisVL supports multiple text scoring algorithms. Let's compare BM25STD and TFIDF:

```
# BM25 standard scoring (default)
bm25_query = TextQuery(
    text="comfortable shoes",
    text_field_name="brief_description",
    text_scorer="BM25STD",
```

(continues on next page)

(continued from previous page)

```

    return_fields=["product_id", "brief_description", "price"],
    num_results=3
)

print("Results with BM25 scoring:")
results = index.query(bm25_query)
result_print(results)

```

Results with BM25 scoring:

<IPython.core.display.HTML object>

```

# TFIDF scoring
tfidf_query = TextQuery(
    text="comfortable shoes",
    text_field_name="brief_description",
    text_scorer="TFIDF",
    return_fields=["product_id", "brief_description", "price"],
    num_results=3
)

print("Results with TFIDF scoring:")
results = index.query(tfidf_query)
result_print(results)

```

Results with TFIDF scoring:

<IPython.core.display.HTML object>

Text Search with Filters

Combine text search with filters to narrow results:

```

from redisvl.query.filter import Tag, Num

# Search for "shoes" only in the footwear category
filtered_text_query = TextQuery(
    text="shoes",
    text_field_name="brief_description",
    filter_expression=Tag("category") == "footwear",
    return_fields=["product_id", "brief_description", "category", "price"],
    num_results=5
)

results = index.query(filtered_text_query)
result_print(results)

```

<IPython.core.display.HTML object>

```
# Search for products under $100
price_filtered_query = TextQuery(
    text="comfortable",
    text_field_name="brief_description",
    filter_expression=Num("price") < 100,
    return_fields=["product_id", "brief_description", "price"],
    num_results=5
)

results = index.query(price_filtered_query)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

Text Search with Multiple Fields and Weights

You can search across multiple text fields with different weights to prioritize certain fields. Here we'll prioritize the `brief_description` field and make text similarity in that field twice as important as text similarity in `full_description`:

```
weighted_query = TextQuery(
    text="shoes",
    text_field_name={"brief_description": 1.0, "full_description": 0.5},
    return_fields=["product_id", "brief_description"],
    num_results=3
)

results = index.query(weighted_query)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

Text Search with Custom Stopwords

Stopwords are common words that are filtered out before processing the query. You can specify which language's default stopwords should be filtered out, like `english`, `french`, or `german`. You can also define your own list of stopwords:

```
# Use English stopwords (default)
query_with_stopwords = TextQuery(
    text="the best shoes for running",
    text_field_name="brief_description",
    stopwords="english", # Common words like "the", "for" will be removed
    return_fields=["product_id", "brief_description"],
    num_results=3
)

results = index.query(query_with_stopwords)
result_print(results)
```

<IPython.core.display.HTML object>

```
# Use custom stopwords
custom_stopwords_query = TextQuery(
    text="professional equipment for athletes",
    text_field_name="brief_description",
    stopwords=["for", "with"], # Only these words will be filtered
    return_fields=["product_id", "brief_description"],
    num_results=3
)

results = index.query(custom_stopwords_query)
result_print(results)
```

<IPython.core.display.HTML object>

```
# No stopwords
no_stopwords_query = TextQuery(
    text="the best shoes for running",
    text_field_name="brief_description",
    stopwords=None, # All words will be included
    return_fields=["product_id", "brief_description"],
    num_results=3
)

results = index.query(no_stopwords_query)
result_print(results)
```

<IPython.core.display.HTML object>

2. Hybrid Queries: Combining Text and Vector Search

Hybrid queries combine text search and vector similarity to provide the best of both worlds:

- **Text search:** Finds exact keyword matches
- **Vector search:** Captures semantic similarity

As of Redis 8.4.0, Redis natively supports a `FT.HYBRID` search command. RedisVL provides a `HybridQuery` class that makes it easy to construct and execute hybrid queries. For earlier versions of Redis, RedisVL provides an `AggregateHybridQuery` class that uses Redis aggregation to achieve similar results.

```
import warnings

# redis-py's hybrid query helpers warn that APIs are experimental; keep notebook output
↳ readable
warnings.filterwarnings("ignore", message=r".*is an experimental.*",
↳ category=UserWarning)

from packaging.version import Version

from redis import __version__ as _redis_py_version
```

(continues on next page)

(continued from previous page)

```

redis_py_version = Version(_redis_py_version)
redis_version = Version(index.client.info()["redis_version"])

HYBRID_SEARCH_AVAILABLE = redis_version >= Version("8.4.0") and redis_py_version >=
↪ Version("7.1.0")
print(HYBRID_SEARCH_AVAILABLE)

```

True

Index-Level Stopwords Configuration

The previous example showed **query-time stopwords** using `TextQuery.stopwords`, which filters words from the query before searching. RedisVL also supports **index-level stopwords** configuration, which determines which words are indexed in the first place.

Key Difference:

- **Query-time stopwords** (`TextQuery.stopwords`): Filters words from your search query (client-side)
- **Index-level stopwords** (`IndexInfo.stopwords`): Controls which words get indexed in Redis (server-side)

Three Configuration Modes:

1. **None (default)**: Use Redis's default stopwords list
2. **[] (empty list)**: Disable stopwords completely (`STOPWORDS 0` in `FT.CREATE`)
3. **["the", "a", "an"]**: Use a custom stopwords list

When to use `STOPWORDS 0`:

- When you need to search for common words like “of”, “at”, “the”
- For entity names containing stopwords (e.g., “Bank of Glasberliner”, “University of Glasberliner”)
- When working with structured data where every word matters

```

# Create a schema with index-level stopwords disabled
from redisvl.index import SearchIndex

stopwords_schema = {
    "index": {
        "name": "company_index",
        "prefix": "company:",
        "storage_type": "hash",
        "stopwords": [] # STOPWORDS 0 - disable stopwords completely
    },
    "fields": [
        {"name": "company_name", "type": "text"},
        {"name": "description", "type": "text"}
    ]
}

# Create index using from_dict (handles schema creation internally)
company_index = SearchIndex.from_dict(stopwords_schema, redis_url="redis://localhost:6379

```

(continues on next page)

(continued from previous page)

```

↪")
company_index.create(overwrite=True, drop=True)

print(f"Index created with STOPWORDS 0: {company_index}")

```

```
Index created with STOPWORDS 0: <redisvl.index.index.SearchIndex object at 0x130e98410>
```

```

# Load sample data with company names containing common stopwords
companies = [
    {"company_name": "Bank of Glasberliner", "description": "Major financial institution"},
    ↪},
    {"company_name": "University of Glasberliner", "description": "Public university"},
    ↪system"},
    {"company_name": "Department of Glasberliner Affairs", "description": "A government"},
    ↪agency"},
    {"company_name": "Glasberliner FC", "description": "Football Club"},
    {"company_name": "The Home Market", "description": "Home improvement retailer"},
]

for i, company in enumerate(companies):
    company_index.load([company], keys=[f"company:{i}"])

print(f"✓ Loaded {len(companies)} companies")

```

```
✓ Loaded 5 companies
```

```

# Search for "Bank of Glasberliner" - with STOPWORDS 0, "of" is indexed and searchable
from redisvl.query import FilterQuery

query = FilterQuery(
    filter_expression="@company_name:(Bank of Glasberliner)",
    return_fields=["company_name", "description"],
)

results = company_index.search(query.query, query_params=query.params)

print(f"Found {len(results.docs)} results for 'Bank of Glasberliner':")
for doc in results.docs:
    print(f"  - {doc.company_name}: {doc.description}")

```

```
Found 1 results for 'Bank of Glasberliner':
  - Bank of Glasberliner: Major financial institution
```

Comparison: With vs Without Stopwords

If we had used the default stopwords (not specifying stopwords in the schema), the word “of” would be filtered out during indexing. This means:

- Searching for "Bank of Glasberliner" might not find exact matches
- The phrase would be indexed as "Bank Berlin" (without “of”)
- With STOPWORDS 0, all words including “of” are indexed

Custom Stopwords Example:

You can also provide a custom list of stopwords:

```
# Example: Create index with custom stopwords
custom_stopwords_schema = {
    "index": {
        "name": "custom_stopwords_index",
        "prefix": "custom:",
        "stopwords": ["inc", "llc", "corp"] # Filter out legal entity suffixes
    },
    "fields": [
        {"name": "name", "type": "text"}
    ]
}

# This would create an index where "inc", "llc", "corp" are not indexed
print("Custom stopwords:", custom_stopwords_schema["index"]["stopwords"])
```

```
Custom stopwords: ['inc', 'llc', 'corp']
```

YAML Format:

You can also define stopwords in YAML schema files:

```
version: '0.1.0'

index:
  name: company_index
  prefix: company:
  storage_type: hash
  stopwords: [] # Disable stopwords (STOPWORDS 0)

fields:
  - name: company_name
    type: text
  - name: description
    type: text
```

Or with custom stopwords:

```
index:
  stopwords:
    - the
    - a
    - an
```

```
# Cleanup
company_index.delete(drop=True)
print("✓ Cleaned up company_index")
```

```
✓ Cleaned up company_index
```

Basic Hybrid Query

NOTE: HybridQuery requires Redis $\geq 8.4.0$ and redis-py $\geq 7.1.0$.

Let's search for "running" with both text and semantic search, combining the results' scores using a linear combination:

```
if HYBRID_SEARCH_AVAILABLE:
    from redisvl.query import HybridQuery

    # Create a hybrid query
    hybrid_query = HybridQuery(
        text="running shoes",
        text_field_name="brief_description",
        vector=[0.1, 0.2, 0.1], # Query vector
        vector_field_name="text_embedding",
        return_fields=["product_id", "brief_description", "category", "price"],
        num_results=5,
        yield_text_score_as="text_score",
        yield_vsim_score_as="vector_similarity",
        combination_method="LINEAR",
        yield_combined_score_as="hybrid_score",
    )

    results = index.query(hybrid_query)
    result_print(results)

else:
    print("Hybrid search is not available in this version of Redis/redis-py.")
```

<IPython.core.display.HTML object>

For earlier versions of Redis, you can use AggregateHybridQuery instead:

```
from redisvl.query import AggregateHybridQuery

agg_hybrid_query = AggregateHybridQuery(
    text="running shoes",
    text_field_name="brief_description",
    vector=[0.1, 0.2, 0.1], # Query vector
    vector_field_name="text_embedding",
    return_fields=["product_id", "brief_description", "category", "price"],
    num_results=5
)

results = index.query(agg_hybrid_query)
result_print(results)
```

<IPython.core.display.HTML object>

Adjusting the Alpha Parameter

Results are scored using a weighted combination:

```
hybrid_score = (alpha) * text_score + (1 - alpha) * vector_score
```

Where alpha controls the balance between text and vector search (default: 0.3 for HybridQuery and 0.7 for AggregateHybridQuery). Note that AggregateHybridQuery reverses the definition of alpha to be the weight of the vector score.

The alpha parameter controls the weight between text and vector search:

- alpha=1.0: Pure text search (or pure vector search for AggregateHybridQuery)
- alpha=0.0: Pure vector search (or pure text search for AggregateHybridQuery)
- alpha=0.3 (default - HybridQuery): 30% text, 70% vector

```
if HYBRID_SEARCH_AVAILABLE:
    vector_heavy_query = HybridQuery(
        text="comfortable",
        text_field_name="brief_description",
        vector=[0.15, 0.25, 0.15],
        vector_field_name="text_embedding",
        combination_method="LINEAR",
        linear_alpha=0.1, # 10% text, 90% vector
        return_fields=["product_id", "brief_description"],
        num_results=3,
        yield_text_score_as="text_score",
        yield_vsim_score_as="vector_similarity",
        yield_combined_score_as="hybrid_score",
    )

    print("Results with alpha=0.1 (vector-heavy):")
    results = index.query(vector_heavy_query)
    result_print(results)

else:
    print("Hybrid search is not available in this version of Redis/redis-py.")
```

Results with alpha=0.1 (vector-heavy):

<IPython.core.display.HTML object>

```
# More emphasis on vector search (alpha=0.9)
vector_heavy_query = AggregateHybridQuery(
    text="comfortable",
    text_field_name="brief_description",
    vector=[0.15, 0.25, 0.15],
    vector_field_name="text_embedding",
    alpha=0.9, # 90% vector, 10% text
    return_fields=["product_id", "brief_description"],
    num_results=3
)
```

(continues on next page)

(continued from previous page)

```
print("Results with alpha=0.9 (vector-heavy):")
results = index.query(vector_heavy_query)
result_print(results)
```

```
Results with alpha=0.9 (vector-heavy):
```

```
<IPython.core.display.HTML object>
```

Reciprocal Rank Fusion (RRF)

In addition to combining scores using a linear combination, `HybridQuery` also supports reciprocal rank fusion (RRF) for combining scores. This method is useful when you want to combine scores giving more weight to the top results from each query.

`HybridQuery` allows for the following parameters to be specified for RRF:

- `rrf_window`: The window size to use for the RRF combination method. Limits the fusion scope.
- `rrf_constant`: The constant to use for the RRF combination method. Controls the decay of rank influence.

`AggregateHybridQuery` does not support RRF, and only supports a linear combination of scores.

```
if HYBRID_SEARCH_AVAILABLE:
    rrf_query = HybridQuery(
        text="comfortable",
        text_field_name="brief_description",
        vector=[0.15, 0.25, 0.15],
        vector_field_name="text_embedding",
        combination_method="RRF",
        return_fields=["product_id", "brief_description"],
        num_results=3,
        yield_text_score_as="text_score",
        yield_vsim_score_as="vector_similarity",
        yield_combined_score_as="hybrid_score",
    )

    results = index.query(rrf_query)
    result_print(results)

else:
    print("Hybrid search is not available in this version of Redis/redis-py.")
```

```
<IPython.core.display.HTML object>
```

Hybrid Query with Filters

You can also combine hybrid search with filters:

```
if HYBRID_SEARCH_AVAILABLE:
    # Hybrid search with a price filter
    filtered_hybrid_query = HybridQuery(
        text="professional equipment",
        text_field_name="brief_description",
        vector=[0.9, 0.1, 0.05],
        vector_field_name="text_embedding",
        filter_expression=Num("price") > 100,
        return_fields=["product_id", "brief_description", "category", "price"],
        num_results=5,
        combination_method="LINEAR",
        yield_text_score_as="text_score",
        yield_vsim_score_as="vector_similarity",
        yield_combined_score_as="hybrid_score",
    )

    results = index.query(filtered_hybrid_query)
    result_print(results)

else:
    print("Hybrid search is not available in this version of Redis/redis-py.")
```

<IPython.core.display.HTML object>

```
# Hybrid search with a price filter
filtered_hybrid_query = AggregateHybridQuery(
    text="professional equipment",
    text_field_name="brief_description",
    vector=[0.9, 0.1, 0.05],
    vector_field_name="text_embedding",
    filter_expression=Num("price") > 100,
    return_fields=["product_id", "brief_description", "category", "price"],
    num_results=5
)

results = index.query(filtered_hybrid_query)
result_print(results)
```

<IPython.core.display.HTML object>

Using Different Text Scorers

Hybrid queries support the same text scoring algorithms as TextQuery:

```
if HYBRID_SEARCH_AVAILABLE:
    # Aggregate Hybrid query with TFIDF scorer
    hybrid_tfidf = HybridQuery(
        text="shoes support",
        text_field_name="brief_description",
        vector=[0.12, 0.18, 0.12],
        vector_field_name="text_embedding",
        text_scorer="TFIDF",
        return_fields=["product_id", "brief_description"],
        num_results=3,
        combination_method="LINEAR",
        yield_text_score_as="text_score",
        yield_vsim_score_as="vector_similarity",
        yield_combined_score_as="hybrid_score",
    )

    results = index.query(hybrid_tfidf)
    result_print(results)

else:
    print("Hybrid search is not available in this version of Redis/redis-py.")
```

<IPython.core.display.HTML object>

```
# Aggregate Hybrid query with TFIDF scorer
hybrid_tfidf = AggregateHybridQuery(
    text="shoes support",
    text_field_name="brief_description",
    vector=[0.12, 0.18, 0.12],
    vector_field_name="text_embedding",
    text_scorer="TFIDF",
    return_fields=["product_id", "brief_description"],
    num_results=3
)

results = index.query(hybrid_tfidf)
result_print(results)
```

<IPython.core.display.HTML object>

Runtime Parameters for Vector Search Tuning

Important: AggregateHybridQuery uses FT.AGGREGATE commands which do NOT support runtime parameters.

Runtime parameters (such as `ef_runtime` for HNSW indexes or `search_window_size` for SVS-VAMANA indexes) are only supported with FT.SEARCH (and partially FT.HYBRID) commands.

For runtime parameter support, use HybridQuery, VectorQuery, or VectorRangeQuery instead:

- HybridQuery: Supports `ef_runtime` for HNSW indexes
- VectorQuery: Supports all runtime parameters (HNSW and SVS-VAMANA)
- VectorRangeQuery: Supports all runtime parameters (HNSW and SVS-VAMANA)
- AggregateHybridQuery: Does NOT support runtime parameters (uses FT.AGGREGATE)

See the **Runtime Parameters** section earlier in this notebook for examples of using runtime parameters with VectorQuery.

3. MultiVectorQuery: Multi-Vector Search

The MultiVectorQuery allows you to search over multiple vector fields simultaneously. This is useful when you have different types of embeddings (e.g., text and image embeddings) and want to find results that match across multiple modalities.

The final score is calculated as a weighted combination:

```
combined_score = w_1 * score_1 + w_2 * score_2 + w_3 * score_3 + ...
```

Basic Multi-Vector Query

First, we need to import the Vector class to define our query vectors:

```
from redisvl.query import MultiVectorQuery, Vector

# Define multiple vectors for the query
text_vector = Vector(
    vector=[0.1, 0.2, 0.1],
    field_name="text_embedding",
    dtype="float32",
    weight=0.7 # 70% weight for text embedding
)

image_vector = Vector(
    vector=[0.8, 0.1],
    field_name="image_embedding",
    dtype="float32",
    weight=0.3 # 30% weight for image embedding
)

# Create a multi-vector query
multi_vector_query = MultiVectorQuery(
    vectors=[text_vector, image_vector],
    return_fields=["product_id", "brief_description", "category"],
```

(continues on next page)

(continued from previous page)

```
    num_results=5
)

results = index.query(multi_vector_query)
result_print(results)
```

```
<IPython.core.display.HTML object>
```

Adjusting Vector Weights

You can adjust the weights to prioritize different vector fields:

```
# More emphasis on image similarity
text_vec = Vector(
    vector=[0.9, 0.1, 0.05],
    field_name="text_embedding",
    dtype="float32",
    weight=0.2 # 20% weight
)

image_vec = Vector(
    vector=[0.1, 0.9],
    field_name="image_embedding",
    dtype="float32",
    weight=0.8 # 80% weight
)

image_heavy_query = MultiVectorQuery(
    vectors=[text_vec, image_vec],
    return_fields=["product_id", "brief_description", "category"],
    num_results=3
)

print("Results with emphasis on image similarity:")
results = index.query(image_heavy_query)
result_print(results)
```

```
Results with emphasis on image similarity:
```

```
<IPython.core.display.HTML object>
```

Multi-Vector Query with Filters

Combine multi-vector search with filters to narrow results:

```
# Multi-vector search with category filter
text_vec = Vector(
    vector=[0.1, 0.2, 0.1],
    field_name="text_embedding",
    dtype="float32",
    weight=0.6
)

image_vec = Vector(
    vector=[0.8, 0.1],
    field_name="image_embedding",
    dtype="float32",
    weight=0.4
)

filtered_multi_query = MultiVectorQuery(
    vectors=[text_vec, image_vec],
    filter_expression=Tag("category") == "footwear",
    return_fields=["product_id", "brief_description", "category", "price"],
    num_results=5
)

results = index.query(filtered_multi_query)
result_print(results)
```

<IPython.core.display.HTML object>

Comparing Query Types

Let's compare the three query types side by side:

```
# TextQuery - keyword-based search
text_q = TextQuery(
    text="shoes",
    text_field_name="brief_description",
    return_fields=["product_id", "brief_description"],
    num_results=3
)

print("TextQuery Results (keyword-based):")
result_print(index.query(text_q))
print()
```

TextQuery Results (keyword-based):

<IPython.core.display.HTML object>

```

if HYBRID_SEARCH_AVAILABLE:
    # HybridQuery - combines text and vector search
    hybrid_q = HybridQuery(
        text="shoes",
        text_field_name="brief_description",
        vector=[0.1, 0.2, 0.1],
        vector_field_name="text_embedding",
        return_fields=["product_id", "brief_description"],
        num_results=3,
        combination_method="LINEAR",
        yield_text_score_as="text_score",
        yield_vsim_score_as="vector_similarity",
        yield_combined_score_as="hybrid_score",
    )

    results = index.query(hybrid_q)

else:
    hybrid_q = AggregateHybridQuery(
        text="shoes",
        text_field_name="brief_description",
        vector=[0.1, 0.2, 0.1],
        vector_field_name="text_embedding",
        return_fields=["product_id", "brief_description"],
        num_results=3,
    )

    results = index.query(hybrid_q)

print(f"{hybrid_q.__class__.__name__} Results (text + vector):")
result_print(results)
print()

```

```
HybridQuery Results (text + vector):
```

```
<IPython.core.display.HTML object>
```

```

# MultiVectorQuery - searches multiple vector fields
mv_text = Vector(
    vector=[0.1, 0.2, 0.1],
    field_name="text_embedding",
    dtype="float32",
    weight=0.5
)

mv_image = Vector(

```

(continues on next page)

(continued from previous page)

```
vector=[0.8, 0.1],
field_name="image_embedding",
dtype="float32",
weight=0.5
)

multi_q = MultiVectorQuery(
    vectors=[mv_text, mv_image],
    return_fields=["product_id", "brief_description"],
    num_results=3
)

print("MultiVectorQuery Results (multiple vectors):")
result_print(index.query(multi_q))
```

```
MultiVectorQuery Results (multiple vectors):
```

```
<IPython.core.display.HTML object>
```

Best Practices

When to Use Each Query Type:

1. **TextQuery:**

- When you need precise keyword matching
- For traditional search engine functionality
- When text relevance scoring is important
- Example: Product search, document retrieval

2. **HybridQuery:**

- When you want to combine keyword and semantic search
- For improved search quality over pure text or vector search
- When you have both text and vector representations of your data
- Example: E-commerce search, content recommendation

3. **MultiVectorQuery:**

- When you have multiple types of embeddings (text, image, audio, etc.)
- For multi-modal search applications
- When you want to balance multiple semantic signals
- Example: Image-text search, cross-modal retrieval

Next Steps

Now that you understand advanced query types, explore these related guides:

- [Query and Filter Data](#) - Apply filters to narrow down search results
- [Write SQL Queries for Redis](#) - Use SQL-like syntax for Redis queries
- [Improve Search Quality with Rerankers](#) - Rerank results for better relevance

Cleanup

```
# Cleanup
index.delete()
```

Write SQL Queries for Redis

While Redis does not natively support SQL, RedisVL provides a `SQLQuery` class that translates SQL-like queries into Redis queries.

The `SQLQuery` class wraps the `sql-redis` package. This package is not installed by default, so install it with:

```
pip install redisvl[sql-redis]
```

Prerequisites

Before you begin, ensure you have:

- Installed RedisVL with SQL support: `pip install redisvl[sql-redis]`
- A running Redis instance ([Redis 8+](#) or [Redis Cloud](#))

What You'll Learn

By the end of this guide, you will be able to:

- Write SQL-like queries for Redis using `SQLQuery`
- Translate `SELECT`, `WHERE`, and `ORDER BY` clauses to Redis queries
- Combine SQL queries with vector search
- Use aggregate functions and grouping
- Query geographic data with `geo_distance()`
- Filter and extract date/time data with `YEAR()`, `MONTH()`, and `DATE_FORMAT()`

Table of Contents

1. *Define the schema*
2. *Create sample dataset*
3. *Create a SearchIndex*
4. *Load data*
5. *Write SQL queries*
6. *Query types*
 - *Text searches*
 - *Aggregations*
 - *Vector search*
 - *Geographic queries*
 - *Date and datetime queries*
7. *Async support*
8. *Additional query examples*
9. *Cleanup*

Define the schema

```
from redisvl.utils.vectorize import HFTextVectorizer

hf = HFTextVectorizer()

schema = {
    "index": {
        "name": "user_simple",
        "prefix": "user_simple_docs",
        "storage_type": "json",
    },
    "fields": [
        {"name": "user", "type": "tag"},
        {"name": "region", "type": "tag"},
        {"name": "job", "type": "tag"},
        {"name": "job_description", "type": "text"},
        {"name": "age", "type": "numeric"},
        {"name": "office_location", "type": "geo"},
        {
            "name": "job_embedding",
            "type": "vector",
            "attrs": {
                "dims": len(hf.embed("get embed length")),
                "distance_metric": "cosine",
                "algorithm": "flat",
                "datatype": "float32"
            }
        }
    ]
}
```

(continues on next page)

(continued from previous page)

```

    }
  ]
}

```

```

/Users/robert.shelton/Documents/redis-vl-python/.venv/lib/python3.11/site-packages/tqdm/
→auto.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets.
→See https://ipywidgets.readthedocs.io/en/stable/user_install.html
from .autonotebook import tqdm as notebook_tqdm

```

Create sample dataset

```

# Office locations use "longitude,latitude" format (lon,lat - Redis convention)
# San Francisco: -122.4194, 37.7749
# Chicago: -87.6298, 41.8781
# New York: -73.9857, 40.7580
data = [
    {
        'user': 'john',
        'age': 34,
        'job': 'software engineer',
        'region': 'us-west',
        'job_description': 'Designs, develops, and maintains software applications and
→systems.',
        'office_location': '-122.4194,37.7749' # San Francisco
    },
    {
        'user': 'bill',
        'age': 54,
        'job': 'engineer',
        'region': 'us-central',
        'job_description': 'Applies scientific and mathematical principles to solve
→technical problems.',
        'office_location': '-87.6298,41.8781' # Chicago
    },
    {
        'user': 'mary',
        'age': 24,
        'job': 'doctor',
        'region': 'us-central',
        'job_description': 'Diagnoses and treats illnesses, injuries, and other medical
→conditions in the healthcare field.',
        'office_location': '-87.6298,41.8781' # Chicago
    },
    {
        'user': 'joe',
        'age': 27,
        'job': 'dentist',
        'region': 'us-east',
        'job_description': 'Provides oral healthcare including diagnosing and treating
→teeth and gum issues.',

```

(continues on next page)

(continued from previous page)

```

        'office_location': '-73.9857,40.7580' # New York
    },
    {
        'user': 'stacy',
        'age': 61,
        'job': 'project manager',
        'region': 'us-west',
        'job_description': 'Plans, organizes, and oversees projects from inception to_
↪completion.',
        'office_location': '-122.4194,37.7749' # San Francisco
    }
]

data = [
    {
        **d,
        "job_embedding": hf.embed(f"{d['job_description']} {d['job']}"),
    }
    for d in data
]

```

Create a SearchIndex

With the schema and sample dataset ready, create a SearchIndex.

Bring your own Redis connection instance

This is ideal in scenarios where you have custom settings on the connection instance or if your application will share a connection pool:

```

from redisvl.index import SearchIndex
from redis import Redis

client = Redis.from_url("redis://localhost:6379")
index = SearchIndex.from_dict(schema, redis_client=client)

```

Let the index manage the connection instance

This is ideal for simple cases:

```

index = SearchIndex.from_dict(schema, redis_url="redis://localhost:6379", validate_on_
↪load=True)

```

Create the index

Now that we are connected to Redis, we need to run the create command.

```
index.create(overwrite=True, drop=True)
```

Load data

Load the sample dataset to Redis.

Validate data entries on load

RedisVL uses pydantic validation under the hood to ensure loaded data is valid and conforms to your schema. This setting is optional and can be configured via `validate_on_load=True` in the `SearchIndex` class.

```
keys = index.load(data)
```

```
print(keys)
```

```
['user_simple_docs:01KN7Y4J630537VY4Y5D9EZMYX', 'user_simple_
↪docs:01KN7Y4J630537VY4Y5D9EZMYX', 'user_simple_docs:01KN7Y4J630537VY4Y5D9EZMYZ', 'user_
↪simple_docs:01KN7Y4J630537VY4Y5D9EZMZ0', 'user_simple_docs:01KN7Y4J630537VY4Y5D9EZMZ1']
```

Write SQL queries

First, let's test a simple select statement such as the one below.

```
from redisvl.query import SQLQuery

sql_str = """
    SELECT user, region, job, age
    FROM user_simple
    WHERE age > 17
    """

# Optional sql_redis_options are passed through to sql-redis.
# schema_cache_strategy balances startup cost vs repeated-query speed:
# use "lazy" (default) to load schemas on demand, or "load_all"
# to preload schemas up front for broader repeated-query workloads.
sql_query = SQLQuery(
    sql_str, sql_redis_options={"schema_cache_strategy": "lazy"}
)
```

Check the created query string

```
sql_query.redis_query_string(redis_url="redis://localhost:6379")
```

```
'FT.SEARCH user_simple "@age:[(17 +inf]" RETURN 4 user region job age DIALECT 2'
```

Executing the query

```
results = index.query(sql_query)
results
```

```
[{'user': 'john',
  'region': 'us-west',
  'job': 'software engineer',
  'age': '34'},
 {'user': 'bill', 'region': 'us-central', 'job': 'engineer', 'age': '54'},
 {'user': 'mary', 'region': 'us-central', 'job': 'doctor', 'age': '24'},
 {'user': 'joe', 'region': 'us-east', 'job': 'dentist', 'age': '27'},
 {'user': 'stacy', 'region': 'us-west', 'job': 'project manager', 'age': '61'}]
```

Query types

Conditional operators

```
sql_str = """
    SELECT user, region, job, age
    FROM user_simple
    WHERE age > 17 and region = 'us-west'
    """

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```

```
Resulting redis query: FT.SEARCH user_simple "@age:[(17 +inf] @region:{us\-west}"
↪RETURN 4 user region job age DIALECT 2
```

```
[{'user': 'john',
  'region': 'us-west',
  'job': 'software engineer',
  'age': '34'},
 {'user': 'stacy', 'region': 'us-west', 'job': 'project manager', 'age': '61'}]
```

```
sql_str = """
    SELECT user, region, job, age
```

(continues on next page)

(continued from previous page)

```
FROM user_simple
WHERE region = 'us-west' or region = 'us-central'
"""
```

```
sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```

```
Resulting redis query: FT.SEARCH user_simple "((@region:{us\-west})|(@region:{us\-
↪central}))" RETURN 4 user region job age
```

```
[{'user': 'mary', 'region': 'us-central', 'job': 'doctor', 'age': '24'},
{'user': 'bill', 'region': 'us-central', 'job': 'engineer', 'age': '54'},
{'user': 'stacy', 'region': 'us-west', 'job': 'project manager', 'age': '61'},
{'user': 'john',
 'region': 'us-west',
 'job': 'software engineer',
 'age': '34'}]
```

job is a tag field therefore this syntax works

```
sql_str = """
SELECT user, region, job, age
FROM user_simple
WHERE job IN ('software engineer', 'engineer', 'pancake tester')
"""
```

```
sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```

```
Resulting redis query: FT.SEARCH user_simple "@job:{software engineer|engineer|pancake_
↪tester}" RETURN 4 user region job age
```

```
[{'user': 'bill', 'region': 'us-central', 'job': 'engineer', 'age': '54'},
{'user': 'john',
 'region': 'us-west',
 'job': 'software engineer',
 'age': '34'}]
```


Text searches

See [the docs](#) for available text queries in Redis.

For more on exact matching see [here](#).

With `sql-redis >= 0.4.0`, TEXT search operators are explicit:

- `WHERE job_description = 'healthcare including'` for exact phrase matching
- `WHERE job_description LIKE 'sci%', LIKE '%care', or LIKE '%diagnose%'` for wildcard matching
- `WHERE fuzzy(job_description, 'diagnose')` for typo-tolerant matching
- `WHERE fulltext(job_description, 'healthcare OR diagnosing')` for tokenized search

```
# Prefix (LIKE)
sql_str = """
    SELECT user, region, job, job_description, age
    FROM user_simple
    WHERE job_description LIKE 'sci%'
"""

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```

```
Resulting redis query: FT.SEARCH user_simple "@job_description:sci*" RETURN 5 user_
↪region job job_description age
```

```
[{'user': 'bill',
  'region': 'us-central',
  'job': 'engineer',
  'job_description': 'Applies scientific and mathematical principles to solve technical_
↪problems.',
  'age': '54'}]
```

```
# Suffix (LIKE)
sql_str = """
    SELECT user, region, job, job_description, age
    FROM user_simple
    WHERE job_description LIKE '%care'
"""

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```

```
Resulting redis query: FT.SEARCH user_simple "@job_description:*care" RETURN 5 user_
↪region job job_description age
```

```
[{'user': 'mary',
  'region': 'us-central',
  'job': 'doctor',
  'job_description': 'Diagnoses and treats illnesses, injuries, and other medical_
↳conditions in the healthcare field.',
  'age': '24'},
 {'user': 'joe',
  'region': 'us-east',
  'job': 'dentist',
  'job_description': 'Provides oral healthcare including diagnosing and treating teeth_
↳and gum issues.',
  'age': '27'}]
```

```
# Contains (LIKE)
sql_str = """
    SELECT user, region, job, job_description, age
    FROM user_simple
    WHERE job_description LIKE '%diagnose%'
"""

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```

```
Resulting redis query: FT.SEARCH user_simple "@job_description:*diagnose*" RETURN 5_
↳user region job job_description age
```

```
[{'user': 'mary',
  'region': 'us-central',
  'job': 'doctor',
  'job_description': 'Diagnoses and treats illnesses, injuries, and other medical_
↳conditions in the healthcare field.',
  'age': '24'}]
```

```
# Phrase no stop words
sql_str = """
    SELECT user, region, job, job_description, age
    FROM user_simple
    WHERE job_description = 'healthcare including'
"""

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```

```
Resulting redis query: FT.SEARCH user_simple "@job_description:"healthcare including""_
↳RETURN 5 user region job job_description age
```

```
[{'user': 'joe',
  'region': 'us-east',
  'job': 'dentist',
  'job_description': 'Provides oral healthcare including diagnosing and treating teeth_
↳and gum issues.',
  'age': '27'}]
```

```
# Phrase with stop words (sql-redis strips default stopwords and warns)
sql_str = """
    SELECT user, region, job, job_description, age
    FROM user_simple
    WHERE job_description = 'diagnosing and treating'
    """

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```

```
Resulting redis query: FT.SEARCH user_simple "@job_description:"diagnosing treating""
↳RETURN 5 user region job job_description age
```

```
[{'user': 'joe',
  'region': 'us-east',
  'job': 'dentist',
  'job_description': 'Provides oral healthcare including diagnosing and treating teeth_
↳and gum issues.',
  'age': '27'}]
```

```
sql_str = """
    SELECT user, region, job, age
    FROM user_simple
    WHERE age BETWEEN 40 and 60
    """

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```

```
Resulting redis query: FT.SEARCH user_simple "@age:[40 60]" RETURN 4 user region job age
```

```
[{'user': 'bill', 'region': 'us-central', 'job': 'engineer', 'age': '54'}]
```

Aggregations

See docs for redis supported reducer functions: [docs](#).

```
sql_str = """
SELECT
    user,
    COUNT(age) as count_age,
    COUNT_DISTINCT(age) as count_distinct_age,
    MIN(age) as min_age,
    MAX(age) as max_age,
    AVG(age) as avg_age,
    STDEV(age) as std_age,
    FIRST_VALUE(age) as fist_value_age,
    ARRAY_AGG(age) as to_list_age,
    QUANTILE(age, 0.99) as quantile_age
FROM user_simple
GROUP BY region
"""

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```

```
Resulting redis query: FT.AGGREGATE user_simple "*" LOAD 3 @age @region @user GROUPBY 1_
→@region REDUCE COUNT 0 AS count_age REDUCE COUNT_DISTINCT 1 @age AS count_distinct_age_
→REDUCE MIN 1 @age AS min_age REDUCE MAX 1 @age AS max_age REDUCE AVG 1 @age AS avg_age_
→REDUCE STDDEV 1 @age AS std_age REDUCE FIRST_VALUE 1 @age AS fist_value_age REDUCE_
→TOLIST 1 @age AS to_list_age REDUCE QUANTILE 2 @age 0.99 AS quantile_age
```

```
[{'region': 'us-west',
  'count_age': '2',
  'count_distinct_age': '2',
  'min_age': '34',
  'max_age': '61',
  'avg_age': '47.5',
  'std_age': '19.091883092',
  'fist_value_age': '61',
  'to_list_age': ['34', '61'],
  'quantile_age': '61'},
 {'region': 'us-central',
  'count_age': '2',
  'count_distinct_age': '2',
  'min_age': '24',
  'max_age': '54',
  'avg_age': '39',
  'std_age': '21.2132034356',
  'fist_value_age': '24',
  'to_list_age': ['24', '54'],
  'quantile_age': '54'},
 {'region': 'us-east',
```

(continues on next page)

(continued from previous page)

```
'count_age': '1',
'count_distinct_age': '1',
'min_age': '27',
'max_age': '27',
'avg_age': '27',
'std_age': '0',
'fist_value_age': '27',
'to_list_age': ['27'],
'quantile_age': '27']}]
```

Vector search

```
sql_str = """
    SELECT user, job, job_description, cosine_distance(job_embedding, :vec) AS vector_
    ↪distance
    FROM user_simple
    ORDER BY vector_distance ASC
    """

vec = hf.embed("looking for someone to use base principles to solve problems", as_
    ↪buffer=True)
sql_query = SQLQuery(sql_str, params={"vec": vec})

redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)

results
```

```
Resulting redis query: FT.SEARCH user_simple "*"=>[KNN 10 @job_embedding $vector AS_
    ↪vector_distance]" PARAMS 2 vector $vector DIALECT 2 RETURN 4 user job job_description_
    ↪vector_distance SORTBY vector_distance ASC
```

```
[{'vector_distance': '0.823510587215',
  'user': 'bill',
  'job': 'engineer',
  'job_description': 'Applies scientific and mathematical principles to solve technical_
    ↪problems.'},
 {'vector_distance': '0.965160369873',
  'user': 'john',
  'job': 'software engineer',
  'job_description': 'Designs, develops, and maintains software applications and systems.
    ↪'},
 {'vector_distance': '1.00401353836',
  'user': 'mary',
  'job': 'doctor',
  'job_description': 'Diagnoses and treats illnesses, injuries, and other medical_
    ↪conditions in the healthcare field.'},
 {'vector_distance': '1.0062687397',
  'user': 'stacy',
```

(continues on next page)

(continued from previous page)

```
'job': 'project manager',
'job_description': 'Plans, organizes, and oversees projects from inception to_
↪completion.'},
{'vector_distance': '1.01110625267',
'user': 'joe',
'job': 'dentist',
'job_description': 'Provides oral healthcare including diagnosing and treating teeth_
↪and gum issues.'}]
```

```
sql_str = """
    SELECT user, region, cosine_distance(job_embedding, :vec) AS vector_distance
    FROM user_simple
    WHERE region = 'us-central'
    ORDER BY vector_distance ASC
    """

vec = hf.embed("looking for someone to use base principles to solve problems", as_
↪buffer=True)
sql_query = SQLQuery(sql_str, params={"vec": vec})

redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)

results
```

```
Resulting redis query: FT.SEARCH user_simple "(@region:{us\-central})=>[KNN 10 @job_
↪embedding $vector AS vector_distance]" PARAMS 2 vector $vector DIALECT 2 RETURN 3 user_
↪region vector_distance SORTBY vector_distance ASC
```

```
[{'vector_distance': '0.823510587215', 'user': 'bill', 'region': 'us-central'},
{'vector_distance': '1.00401353836', 'user': 'mary', 'region': 'us-central'}]
```

Hybrid search (FT.HYBRID)

Use `hybrid_vector_search()` to fuse a full-text query and a vector query into a single ranking **server-side** with Redis's native FT.HYBRID. It composes the `cosine_distance()` (vector leg) and `fulltext()` (text leg) functions, with `rrf()` or `linear()` selecting the fusion method (reciprocal rank fusion or a linear weighting).

This differs from the *pre-filter* hybrid search above (a WHERE clause narrowing a KNN search): here both legs are ranked independently and fused, rather than text acting only as a filter.

Requires Redis 8.4+ and `redis-py >= 7.1.0`.

```
sql_str = """
    SELECT user, job, job_description,
           hybrid_vector_search(
               cosine_distance(job_embedding, :vec),
               fulltext(job_description, 'principles solve problems'),
               rrf()
           ) AS hybrid_score
```

(continues on next page)

(continued from previous page)

```

FROM user_simple
ORDER BY hybrid_score DESC
LIMIT 5
"""

vec = hf.embed("looking for someone to use base principles to solve problems", as_
↪buffer=True)
sql_query = SQLQuery(sql_str, params={"vec": vec})

redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)

results

```

Geographic queries

Use `geo_distance()` to filter by location or calculate distances between points.

Syntax:

- Filter: `WHERE geo_distance(field, POINT(lon, lat), 'unit') < radius`
- Distance: `SELECT geo_distance(field, POINT(lon, lat)) AS distance`

Units: 'km' (kilometers), 'mi' (miles), 'm' (meters), 'ft' (feet)

Note: `POINT()` uses longitude first, then latitude - matching Redis conventions.

```

# Find users within 500km of San Francisco
sql_str = """
    SELECT user, job, region, office_location
    FROM user_simple
    WHERE geo_distance(office_location, POINT(-122.4194, 37.7749), 'km') < 500
"""

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)

results

```

```

Resulting redis query: FT.SEARCH user_simple "" GEOFILTER office_location -122.4194 37.
↪7749 500.0 km RETURN 4 user job region office_location

```

```

[{'user': 'stacy',
  'job': 'project manager',
  'region': 'us-west',
  'office_location': '-122.4194,37.7749'},
 {'user': 'john',
  'job': 'software engineer',
  'region': 'us-west',
  'office_location': '-122.4194,37.7749'}]

```

```
# Find users within 50 miles of Chicago (using miles)
sql_str = """
    SELECT user, job, region
    FROM user_simple
    WHERE geo_distance(office_location, POINT(-87.6298, 41.8781), 'mi') < 50
"""

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```

Resulting redis query: FT.SEARCH user_simple "*" GEOFILTER office_location -87.6298 41.8781 50.0 mi RETURN 3 user job region

```
[{'user': 'mary', 'job': 'doctor', 'region': 'us-central'},
 {'user': 'bill', 'job': 'engineer', 'region': 'us-central'}]
```

```
# Combine GEO filter with TAG filter - find engineers near Chicago
sql_str = """
    SELECT user, job, region
    FROM user_simple
    WHERE job = 'engineer' AND geo_distance(office_location, POINT(-87.6298, 41.8781),
    ↪ 'mi') < 50
"""

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```

Resulting redis query: FT.SEARCH user_simple "@job:{engineer}" GEOFILTER office_location -87.6298 41.8781 50.0 mi RETURN 3 user job region

```
[{'user': 'bill', 'job': 'engineer', 'region': 'us-central'}]
```

```
# Combine GEO with NUMERIC filter - find users over 30 near San Francisco
sql_str = """
    SELECT user, job, age
    FROM user_simple
    WHERE age > 30 AND geo_distance(office_location, POINT(-122.4194, 37.7749), 'km') <
    ↪ 100
"""

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```



```
Resulting redis query: FT.SEARCH user_simple "@age:[(30 +inf]" GEOFILTER office_
↳location -122.4194 37.7749 100.0 km RETURN 3 user job age
```

```
[{'user': 'stacy', 'job': 'project manager', 'age': '61'},
 {'user': 'john', 'job': 'software engineer', 'age': '34'}]
```

```
# Combine GEO with TEXT search - find users with "technical" in job description near
↳Chicago
sql_str = """
    SELECT user, job, job_description
    FROM user_simple
    WHERE job_description LIKE 'technical%' AND geo_distance(office_location, POINT(-87.
↳6298, 41.8781), 'km') < 100
    """

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)
results
```

```
Resulting redis query: FT.SEARCH user_simple "@job_description:technical*" GEOFILTER
↳office_location -87.6298 41.8781 100.0 km RETURN 3 user job job_description
```

```
[{'user': 'bill',
 'job': 'engineer',
 'job_description': 'Applies scientific and mathematical principles to solve technical
↳problems.'}]
```

```
# Calculate distances from New York to all users
# Note: geo_distance() in SELECT uses FT.AGGREGATE and returns distance in meters
sql_str = """
    SELECT user, region, geo_distance(office_location, POINT(-73.9857, 40.7580)) AS
↳distance_meters
    FROM user_simple
    """

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = index.query(sql_query)

# Convert meters to km for readability and sort by distance
print("\nDistances from NYC:")
for r in sorted(results, key=lambda x: float(x.get('distance_meters', 0))):
    dist_km = float(r.get('distance_meters', 0)) / 1000
    print(f" {r['user']:10} | {r['region']:12} | {dist_km:,.0f} km")
```

```
Resulting redis query: FT.AGGREGATE user_simple "*" LOAD 3 @office_location @region
↳@user APPLY geodistance(@office_location, -73.9857, 40.758) AS distance_meters
```

(continues on next page)

(continued from previous page)

Distances from NYC:

joe	us-east	0 km
mary	us-central	1,145 km
bill	us-central	1,145 km
stacy	us-west	4,131 km
john	us-west	4,131 km

GEO Query Summary

Method	Pattern	Example
SQL - Basic radius	WHERE geo_distance(field, POINT(lon, lat), 'unit') < radius	WHERE geo_distance(location, POINT(-122.4, 37.8), 'km') < 50
SQL - With miles	Same with 'mi' unit	WHERE geo_distance(location, POINT(-73.9, 40.7), 'mi') < 10
SQL - With TAG	Combined with AND	WHERE category = 'retail' AND geo_distance(...) < 100
SQL - With NUMERIC	Combined with AND	WHERE age > 30 AND geo_distance(...) < 100
SQL - Distance calc	SELECT geo_distance(...)	SELECT geo_distance(location, POINT(lon, lat)) AS dist
Native - Within	Geo(field) == GeoRadius(...)	Geo("location") == GeoRadius(-122.4, 37.8, 100, "km")
Native - Out-side	Geo(field) != GeoRadius(...)	Geo("location") != GeoRadius(-87.6, 41.9, 1000, "km")
Native - Combined	Use & and operators	geo_filter & tag_filter & num_filter

Key Points:

- 1. **Coordinate Format:** "longitude,latitude" - longitude first!
- 2. **POINT() Syntax:** POINT(lon, lat) - longitude first (matches Redis)
- 3. **Units:** 'km', 'mi', 'm', 'ft'
- 4. **geo_distance():** Returns meters, divide by 1000 for km

Date and datetime queries

Use date literals and functions to query timestamp data. Redis stores dates as Unix timestamps in NUMERIC fields.

Key Concepts:

- Date literals like '2024-01-01' are auto-converted to Unix timestamps
- Date functions (YEAR(), MONTH(), DAY()) extract date parts
- DATE_FORMAT() formats timestamps as readable strings

```
# Create a separate index for date examples
from datetime import datetime, timezone
```

(continues on next page)

(continued from previous page)

```

def to_timestamp(date_str):
    """Convert ISO date string to Unix timestamp (UTC)."""
    dt = datetime.strptime(date_str, "%Y-%m-%d")
    dt = dt.replace(tzinfo=timezone.utc)
    return int(dt.timestamp())

# Define schema with NUMERIC fields for timestamps
events_schema = {
    "index": {
        "name": "events",
        "prefix": "event:",
        "storage_type": "hash",
    },
    "fields": [
        {"name": "name", "type": "text", "attrs": {"sortable": True}},
        {"name": "category", "type": "tag", "attrs": {"sortable": True}},
        {"name": "created_at", "type": "numeric", "attrs": {"sortable": True}},
    ],
}

events_index = SearchIndex.from_dict(events_schema, redis_url="redis://localhost:6379")
events_index.create(overwrite=True)

# Sample events spanning 2023-2024
events = [
    {"name": "New Year Kickoff", "category": "meeting", "created_at": to_timestamp("2024-
    ↪01-01")},
    {"name": "Q1 Planning", "category": "meeting", "created_at": to_timestamp("2024-01-15
    ↪")},
    {"name": "Product Launch", "category": "release", "created_at": to_timestamp("2024-
    ↪02-20")},
    {"name": "Team Offsite", "category": "meeting", "created_at": to_timestamp("2024-03-
    ↪10")},
    {"name": "Summer Summit", "category": "conference", "created_at": to_timestamp("2024-
    ↪07-15")},
    {"name": "Holiday Party 2023", "category": "conference", "created_at": to_timestamp(
    ↪"2023-12-15")},
    {"name": "Year End Review 2023", "category": "meeting", "created_at": to_timestamp(
    ↪"2023-12-20")},
]

events_index.load(events)

print(f"Loaded {len(events)} events:")
for e in events:
    date = datetime.fromtimestamp(e["created_at"], tz=timezone.utc).strftime("%Y-%m-%d")
    print(f"  - {e['name']:25} | {date} | {e['category']}")

```

```

Loaded 7 events:
- New Year Kickoff           | 2024-01-01 | meeting
- Q1 Planning                | 2024-01-15 | meeting
- Product Launch             | 2024-02-20 | release

```

(continues on next page)

(continued from previous page)

- Team Offsite	2024-03-10	meeting
- Summer Summit	2024-07-15	conference
- Holiday Party 2023	2023-12-15	conference
- Year End Review 2023	2023-12-20	meeting

```
# Find events after January 1st, 2024 using date literal
sql_str = """
    SELECT name, category
    FROM events
    WHERE created_at > '2024-01-01'
"""

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = events_index.query(sql_query)

print(f"\nEvents after 2024-01-01 ({len(results)} found):")
for r in results:
    print(f"    - {r['name']}")
```

Resulting redis query: FT.SEARCH events "@created_at:[(1704067200 +inf]" RETURN 2 name,
↪category

Events after 2024-01-01 (4 found):

- Summer Summit
- Q1 Planning
- Team Offsite
- Product Launch

```
# Find events in Q1 2024 using BETWEEN
sql_str = """
    SELECT name, category
    FROM events
    WHERE created_at BETWEEN '2024-01-01' AND '2024-03-31'
"""

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = events_index.query(sql_query)

print(f"\nEvents in Q1 2024 ({len(results)} found):")
for r in results:
    print(f"    - {r['name']} ({r['category']})")
```

Resulting redis query: FT.SEARCH events "@created_at:[1704067200 1711843200]" RETURN 2,
↪name category

Events in Q1 2024 (4 found):

- Q1 Planning (meeting)

(continues on next page)

(continued from previous page)

- New Year Kickoff (meeting)
- Team Offsite (meeting)
- Product Launch (release)

```
# Combine date filter with TAG filter - find meetings in H1 2024
sql_str = """
    SELECT name
    FROM events
    WHERE category = 'meeting' AND created_at BETWEEN '2024-01-01' AND '2024-06-30'
"""

sql_query = SQLQuery(sql_str)
results = events_index.query(sql_query)

print(f"Meetings in H1 2024 ({len(results)} found):")
for r in results:
    print(f"    - {r['name']}")
```

Meetings in H1 2024 (3 found):

- Q1 Planning
- New Year Kickoff
- Team Offsite

Date Query Summary

Pattern	Example
After date	WHERE created_at > '2024-01-01'
Before date	WHERE created_at < '2024-12-31'
Date range	WHERE created_at BETWEEN '2024-01-01' AND '2024-03-31'
Extract year	SELECT YEAR(created_at) AS year
Extract month	SELECT MONTH(created_at) AS month (returns 0-11)
Filter by year	WHERE YEAR(created_at) = 2024
Group by date	GROUP BY YEAR(created_at)
Format date	DATE_FORMAT(created_at, '%Y-%m-%d')

Key Points:

1. **Storage:** Dates stored as Unix timestamps in NUMERIC fields
2. **Date Literals:** ISO 8601 strings auto-converted to timestamps
3. **Timezone:** Dates without timezone are treated as UTC
4. **Month Index:** Redis MONTH() returns 0-11, not 1-12

Async support

SQL queries also work with AsyncSearchIndex for async applications:

```
from redisvl.index import AsyncSearchIndex
from redisvl.query import SQLQuery

# Create async index
async_index = AsyncSearchIndex.from_dict(schema, redis_url="redis://localhost:6379")

# Execute SQL query asynchronously
sql_query = SQLQuery(f"SELECT user, age FROM {async_index.name} WHERE age > 30")
results = await async_index.query(sql_query)

# Cleanup
await async_index.disconnect()
```

Additional Query Examples

The following sections provide more detailed examples for geographic and date queries.

Native GEO filters

As an alternative to SQL syntax, RedisVL provides native Geo and GeoRadius filter classes. These can be combined with other filters using & (AND) and | (OR) operators.

```
from redisvl.query import FilterQuery
from redisvl.query.filter import Geo, GeoRadius, Tag, Num

# Find users within 100km of Chicago using native filters
geo_filter = Geo("office_location") == GeoRadius(-87.6298, 41.8781, 100, "km")

print(f"Filter expression: {geo_filter}\n")

query = FilterQuery(
    filter_expression=geo_filter,
    return_fields=["user", "job", "region"]
)

results = index.query(query)
print(f"Users within 100km of Chicago ({len(results)} found):")
for r in results:
    print(f"  - {r['user']} ({r['job']}) - {r['region']}")
```

```
Filter expression: @office_location:[-87.6298 41.8781 100 km]
```

```
Users within 100km of Chicago (2 found):
```

- mary (doctor) - us-central
- bill (engineer) - us-central

```
# Find users OUTSIDE 1000km of Chicago (using !=)
geo_filter_outside = Geo("office_location") != GeoRadius(-87.6298, 41.8781, 1000, "km")

print(f"Filter expression: {geo_filter_outside}\n")

query = FilterQuery(
    filter_expression=geo_filter_outside,
    return_fields=["user", "region"]
)

results = index.query(query)
print(f"Users OUTSIDE 1000km of Chicago ({len(results)} found):")
for r in results:
    print(f"  - {r['user']} ({r['region']})")
```

Filter expression: (-@office_location:[-87.6298 41.8781 1000 km])

Users OUTSIDE 1000km of Chicago (3 found):

- joe (us-east)
- stacy (us-west)
- john (us-west)

```
# Combine GEO + TAG + NUMERIC filters
# Find engineers over 40 within 500km of Chicago
geo_filter = Geo("office_location") == GeoRadius(-87.6298, 41.8781, 500, "km")
job_filter = Tag("job") == "engineer"
age_filter = Num("age") > 40

combined_filter = geo_filter & job_filter & age_filter

print(f"Combined filter: {combined_filter}\n")

query = FilterQuery(
    filter_expression=combined_filter,
    return_fields=["user", "job", "age", "region"]
)

results = index.query(query)
print(f"Engineers over 40 within 500km of Chicago ({len(results)} found):")
for r in results:
    print(f"  - {r['user']} (age: {r['age']}) - {r['region']}")
```

Combined filter: ((@office_location:[-87.6298 41.8781 500 km] @job:{engineer}) @age:[(40,
 ↪+inf]))

Engineers over 40 within 500km of Chicago (1 found):

- bill (age: 54) - us-central

Additional Date Examples

More advanced date query patterns including date function extraction and formatting.

```
# Extract YEAR and MONTH using date functions in SELECT
sql_str = """
    SELECT name, YEAR(created_at) AS year, MONTH(created_at) AS month
    FROM events
    """

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = events_index.query(sql_query)

print(f"\nEvents with year/month:")
for r in results:
    # Note: MONTH returns 0-11 in Redis (0=January)
    month_num = int(r.get('month', 0)) + 1
    print(f" - {r['name']}:25} | {r.get('year')}-{'month_num:02d}'")
```

Resulting redis query: FT.AGGREGATE events "*" LOAD 2 @created_at @name APPLY_↪year(@created_at) AS year APPLY monthofyear(@created_at) AS month

Events with year/month:

- Summer Summit	2024-07
- Q1 Planning	2024-01
- Year End Review 2023	2023-12
- New Year Kickoff	2024-01
- Holiday Party 2023	2023-12
- Team Offsite	2024-03
- Product Launch	2024-02

```
# Filter by YEAR using date function in WHERE
sql_str = """
    SELECT name
    FROM events
    WHERE YEAR(created_at) = 2024
    """

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = events_index.query(sql_query)

print(f"\nEvents in 2024 ({len(results)} found):")
for r in results:
    print(f" - {r['name']}")
```

Resulting redis query: FT.AGGREGATE events "*" LOAD 2 @created_at @name APPLY_↪year(@created_at) AS year_created_at FILTER @year_created_at == 2024

(continues on next page)

(continued from previous page)

Events in 2024 (5 found):

- Summer Summit
- Q1 Planning
- New Year Kickoff
- Team Offsite
- Product Launch

```
# Count events per year using GROUP BY
sql_str = """
    SELECT YEAR(created_at) AS year, COUNT(*) AS event_count
    FROM events
    GROUP BY year
    """

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = events_index.query(sql_query)

print("\nEvents per year:")
for r in sorted(results, key=lambda x: x.get('year', 0)):
    print(f"    {r['year']}: {r['event_count']} events")
```

Resulting redis query: FT.AGGREGATE events "*" LOAD 2 @created_at @year APPLY_
 ↳ year(@created_at) AS year GROUPBY 1 @year REDUCE COUNT 0 AS event_count

Events per year:
 2023: 2 events
 2024: 5 events

```
# Format dates using DATE_FORMAT
sql_str = """
    SELECT name, DATE_FORMAT(created_at, '%Y-%m-%d') AS event_date
    FROM events
    """

sql_query = SQLQuery(sql_str)
redis_query = sql_query.redis_query_string(redis_url="redis://localhost:6379")
print("Resulting redis query: ", redis_query)
results = events_index.query(sql_query)

print("\nEvents with formatted dates:")
for r in results:
    print(f"    - {r['name']}: {r.get('event_date', 'N/A')}")
```

Resulting redis query: FT.AGGREGATE events "*" LOAD 2 @created_at @name APPLY_
 ↳ timefmt(@created_at, "%Y-%m-%d") AS event_date

Events with formatted dates:

- Summer Summit	2024-07-15
- Q1 Planning	2024-01-15

(continues on next page)

(continued from previous page)

- Year End Review 2023	2023-12-20
- New Year Kickoff	2024-01-01
- Holiday Party 2023	2023-12-15
- Team Offsite	2024-03-10
- Product Launch	2024-02-20

Next Steps

Now that you understand SQL queries for Redis, explore these related guides:

- [Use Advanced Query Types](#) - Learn about TextQuery, HybridQuery, and MultiVectorQuery
- [Query and Filter Data](#) - Apply filters using native RedisVL query syntax
- [Getting Started](#) - Review the basics of RedisVL indexes

Cleanup

To remove all data from Redis associated with the index, use the `.clear()` method. This leaves the index in place for future insertions or updates.

To remove everything including the index, use `.delete()` which removes both the index and the underlying data.

```
# Delete both indexes and all associated data
events_index.delete(drop=True)
index.delete(drop=True)
```

Run RedisVL MCP

This guide shows how to run the RedisVL MCP server against an existing Redis index, configure its behavior, and use the MCP tools it exposes.

For the higher-level design, see [RedisVL MCP](#).

Before You Start

RedisVL MCP assumes all of the following are already true:

- you have Python 3.10 or newer
- you have Redis with Search capabilities available
- the Redis index already exists
- you know which text field and vector field the server should use
- you have installed the vectorizer provider dependencies your config needs

Install the MCP extra:

```
pip install redisvl[mcp]
```

If your vectorizer needs a provider extra, install that too:

```
pip install redisvl[mcp,openai]
```

Start the Server

Run the server over stdio (default):

```
uvx --from redisvl[mcp] rvl mcp --config /path/to/mcp.yaml
```

Run it over Streamable HTTP for remote MCP clients. Binding to a non-loopback host (`--host 0.0.0.0`) requires either JWT authentication (see [Authenticate RedisVL MCP](#)) or the explicit `--allow-unauthenticated` flag; otherwise the server refuses to start:

```
uvx --from redisvl[mcp] rvl mcp --config /path/to/mcp.yaml --transport streamable-http --
↪host 0.0.0.0 --port 8000 --allow-unauthenticated
```

Run it over SSE:

```
uvx --from redisvl[mcp] rvl mcp --config /path/to/mcp.yaml --transport sse --host 0.0.0.
↪0 --port 9000 --allow-unauthenticated
```

Warning

Streamable HTTP and SSE endpoints are **unauthenticated by default**. Binding to a non-loopback host without auth fails closed unless you pass `--allow-unauthenticated`; binding to loopback without auth only warns. For real deployments, enable JWT authentication (see [Authenticate RedisVL MCP](#)) rather than using `--allow-unauthenticated`. When not using `--read-only`, the `upsert-records` tool is also exposed to any client that can reach the server.

Transport Security (Host / Origin Validation)

On the HTTP transports the server validates the request `Host` and `Origin` headers against allowlists before any tool runs. This is **on by default** and defends against [DNS rebinding](#): without it, a malicious web page could rebind its own hostname to `127.0.0.1` and use the victim's browser to reach a local MCP server, even one bound to loopback.

- **Host:** the allowlist is derived automatically from the bind address. Loopback binds accept `localhost`, `127.0.0.1`, and `[::1]` (with or without the port), so local MCP clients work with no configuration.
- **Origin:** requests with no `Origin` header (typical of non-browser MCP clients) always pass. A request that carries a cross-site `Origin` is rejected unless that origin is explicitly allowlisted.

You only need to configure this for deployments where the client-visible `Host` differs from the bind address (a reverse proxy, or a `--host 0.0.0.0` bind reached via a public hostname). Use the `REDISVL_MCP_ALLOWED_HOSTS` / `REDISVL_MCP_ALLOWED_ORIGINS` env vars, or a `server.transport_security` block in the config:

```
server:
  redis_url: ${REDIS_URL}
  transport_security:
    allowed_hosts: [mcp.example.com]           # extra Host values to accept
    allowed_origins: [https://app.example.com] # browser origins to accept
    # allow_any_origin: true                  # trust upstream Origin validation
    # enabled: false                         # disable when a trusted proxy validates
```

Note

Behind a reverse proxy or gateway that already validates Host/Origin, set `server.transport_security.enabled: false` (or `allow_any_origin: true`) so the proxy's rewritten headers are not rejected.

Run it in read-only mode to expose search without upsert:

```
uvx --from redisvl[mcp] rvl mcp --config /path/to/mcp.yaml --read-only
```

CLI Flags

Flag	Default	Purpose
<code>--config</code>	—	Path to the MCP YAML config (required)
<code>--transport</code>	<code>stdio</code>	Transport protocol: <code>stdio</code> , <code>sse</code> , or <code>streamable-http</code>
<code>--host</code>	<code>127.0.0.1</code>	Bind address (only used with <code>sse</code> and <code>streamable-http</code>)
<code>--port</code>	<code>8000</code>	Bind port (only used with <code>sse</code> and <code>streamable-http</code>)
<code>--read-only</code>	<code>off</code>	Disable writes across every index (global read-only)

Environment Variables

You can also control boot settings through environment variables:

Variable	Purpose
<code>REDISVL_MCP_CONFIG</code>	Path to the MCP YAML config
<code>REDISVL_MCP_READ_ONLY</code>	Disable <code>upsert</code> -records when set to <code>true</code>
<code>REDISVL_MCP_TOOL_DESCRIPTION</code>	Set the base search tool description text. On a single-index server RedisVL appends schema-derived typed filter, <code>exists</code> , and <code>return_fields</code> hints; on a multi-index server it appends a note directing clients to call <code>list-indexes</code> and pass <code>index</code>
<code>REDISVL_MCP_TOOL_OVERRIDE</code>	Override the upsert tool description
<code>REDISVL_MCP_ALLOWED_HOSTS</code>	Comma-separated extra Host header values to accept on HTTP transports (in addition to the bind-derived defaults)
<code>REDISVL_MCP_ALLOWED_ORIGINS</code>	Comma-separated browser Origin values to accept on HTTP transports
<code>REDISVL_MCP_ALLOW_ANY_ORIGIN</code>	Accept any Origin (for trusted-proxy setups) when set to <code>true</code>
<code>REDISVL_MCP_VALIDATE_HOST_OR_ORIGIN</code>	Set to <code>false</code> to disable Host/Origin validation (e.g. behind a validating proxy)

Connect a Remote MCP Client

When using Streamable HTTP or SSE transport, point your MCP client at the server URL:

- **Streamable HTTP:** `http://<host>:<port>/mcp`
- **SSE:** `http://<host>:<port>/sse`

Note: `<host>` here is the bind address the server was started with. The default `127.0.0.1` only accepts connections from the same machine. To allow connections from other machines, start the server with `--host 0.0.0.0` and use the machine's actual IP or hostname in the client URL. Because a `0.0.0.0` bind has no single canonical Host, add the client-visible hostname(s) to `REDISVL_MCP_ALLOWED_HOSTS`

(or `server.transport_security.allowed_hosts`), or Host validation will reject those requests. See *Transport Security*.

For example, to configure a remote MCP client to connect to a Streamable HTTP server running on 192.168.1.10:8000:

```
{
  "mcpServers": {
    "redisvl": {
      "url": "http://192.168.1.10:8000/mcp",
      "transport": "streamable-http"
    }
  }
}
```

Example Config

This example binds one logical MCP server to one existing Redis index called `knowledge`. A single configured index is the simplest deployment, and callers never need to name it. See *Multiple Indexes* below to expose several indexes from the same server.

The config uses `${REDIS_URL}` and `${OPENAI_API_KEY}` as environment-variable placeholders. These values are resolved when the server starts. You can also use `${VAR:-default}` to provide a fallback value.

```
server:
  redis_url: ${REDIS_URL}

indexes:
  knowledge:
    redis_name: knowledge

    vectorizer:
      class: OpenAITextVectorizer
      model: text-embedding-3-small
      api_config:
        api_key: ${OPENAI_API_KEY}

    schema_overrides:
      fields:
        - name: embedding
          type: vector
          attrs:
            dims: 1536
            datatype: float32

    search:
      type: hybrid
      params:
        text_scorer: BM25STD
        stopwords: english
        vector_search_method: KNN
        combination_method: LINEAR
        linear_text_weight: 0.3
```

(continues on next page)

(continued from previous page)

```
runtime:
  text_field_name: content
  vector_field_name: embedding
  default_embed_text_field: content
  default_limit: 10
  max_limit: 25
  max_result_window: 1000
  max_upsert_records: 64
  skip_embedding_if_present: true
  startup_timeout_seconds: 30
  request_timeout_seconds: 60
  max_concurrency: 16
```

What This Config Means

- `redis_name` must point to an index that already exists in Redis
- `search.type` fixes retrieval behavior for every MCP caller
- `runtime.text_field_name` is required for `fulltext` and `hybrid` search
- `runtime.vector_field_name` is required for `vector` and `hybrid` search, and optional for plain full-text deployments
- `runtime.default_embed_text_field` is only required when the server should generate embeddings during `upsert`
- `vectorizer` is required for query embedding and server-side embedding, but optional for `fulltext`-only configs
- `runtime.max_result_window` caps deep paging by limiting the maximum `offset + limit`
- `schema_overrides` is only for patching incomplete field attrs discovered from Redis

Fulltext-Only Config

For a non-vector deployment, omit vector-only settings entirely:

```
server:
  redis_url: ${REDIS_URL}

indexes:
  knowledge:
    redis_name: knowledge

    search:
      type: fulltext
      params:
        text_scorer: BM25STD
        stopwords: english

    runtime:
      text_field_name: content
```

(continues on next page)

(continued from previous page)

```

default_limit: 10
max_limit: 25
max_result_window: 1000
max_upsert_records: 64
skip_embedding_if_present: true
startup_timeout_seconds: 30
request_timeout_seconds: 60
max_concurrency: 16

```

Multiple Indexes

The indexes mapping can hold more than one binding. Each entry is keyed by a logical id, points at its own existing Redis index through `redis_name`, and carries its own search, runtime, and optional vectorizer. The example below exposes a writable vector index `knowledge` alongside a read-only fulltext index `tickets` from the same server:

```

server:
  redis_url: ${REDIS_URL}

indexes:
  knowledge:
    redis_name: knowledge
    description: Internal runbooks and operational guidance.

    vectorizer:
      class: OpenAITextVectorizer
      model: text-embedding-3-small
      api_config:
        api_key: ${OPENAI_API_KEY}

    search:
      type: vector

    runtime:
      text_field_name: content
      vector_field_name: embedding
      default_embed_text_field: content
      default_limit: 10
      max_limit: 25

  tickets:
    redis_name: support-tickets
    description: Read-only mirror of resolved support tickets.
    read_only: true

    search:
      type: fulltext
      params:
        text_scorer: BM25STD
        stopwords: english

    runtime:

```

(continues on next page)

(continued from previous page)

```

text_field_name: body
default_limit: 10
max_limit: 50

```

Notes:

- Each binding is inspected and validated independently at startup. Startup is all-or-nothing: if any binding fails, the server does not start.
- The optional per-index description and `read_only` flags are surfaced through `list-indexes`.
- `read_only: true` makes that binding reject writes even though the server as a whole is not in global read-only mode. Because `knowledge` is still writable, the `upsert-records` tool is registered; an upsert targeting tickets is rejected with `forbidden`.
- A single-index config keeps working unchanged — adding bindings does not change how the sole-binding case behaves.

Index Selection

On a multi-index server, `search-records` and `upsert-records` take an optional `index` argument naming the logical id to target:

- With exactly one index configured, `index` may be omitted and resolves to that binding.
- With multiple indexes configured, omitting `index` returns `invalid_request`; an unknown id also returns `invalid_request`.
- Clients should call `list-indexes` first to discover the available ids and their filterable fields.

Custom Tool Profiles

A **profile** publishes `search-records` under your own name and description with some arguments pre-filled and frozen. It is pure config — no Python — and it resolves to a built-in call, so it inherits the same limits, read-only policy, auth scoping, and error contract.

```

server:
  redis_url: redis://localhost:6379
  builtin_tools:
    search-records: disabled          # the curated tool replaces it

indexes:
  tickets:
    redis_name: support-tickets
    search:
      type: fulltext
    runtime:
      text_field_name: body
      default_limit: 5
      max_limit: 20

custom_tools:
  - name: search-resolved-tickets
    based_on: search-records

```

(continues on next page)

(continued from previous page)

```

index: tickets
description: >
  Search resolved customer support tickets by relevance.
  Use this to find how a similar problem was fixed before.
lock:
  return_fields: [subject, resolution, created_at]
  filter: { field: status, op: eq, value: resolved }
params:
  limit: { expose: true, max: 10 }
  filter: { expose: true }

```

What the client sees for `search-resolved-tickets`:

- `query` (required), `limit` (10), `offset`, and `filter` — and nothing else.
- No `index` argument: the binding is pinned by `index`: in the config.
- No `return_fields` argument: locking a projection removes it from the contract.
- `filter` accepts the **object** form only. A raw string filter is refused — by the advertised schema for a compliant client, and by the tool itself otherwise.

Rules worth knowing:

- **Anything not listed under `params` stays exposed.** Locking only a filter keeps the rest of the built-in's contract intact.
- **Locked and caller filters are AND-combined.** The caller can narrow within the locked scope but never widen past it — an `or` from the caller nests inside the locked `AND` rather than replacing it.
- **`params.limit.max` bounds the result count either way.** An omitted `limit` is capped rather than falling through to `default_limit`; an explicit request above the cap returns `invalid_request`. With `expose: false`, the cap becomes the fixed result count. The cap may not exceed the binding's `max_limit`. It bounds page size, not total reachable data — `offset` still pages within `max_result_window`.
- **`suppress_schema_hints: true`** drops the auto-generated filter/return-field hints from the description. By default they are appended only for arguments the model can still use.
- **`index` may be omitted only with exactly one index configured.**

`builtin_tools` is independent of profiles: use it to stop advertising any built-in, for example `upsert-records`: disabled on a server that should only read. Disabling `list-indexes` on a multi-index server logs a warning, because `search-records` tells clients to call it to discover index ids.

Misconfiguration fails at startup, not at the first call — a name colliding with a built-in or using a reserved `redisvl-/redisvl_` prefix, a duplicate name, a missing or unknown `index`, a cap above `max_limit`, hiding `query`, locking `return_fields` while also exposing them, or a locked filter or projection naming a field the index does not have. Unrecognized keys are rejected too, so a typo in `lock` fails loudly instead of quietly producing a tool that reads as locked but enforces nothing.

Tool Contracts

RedisVL MCP exposes a small, implementation-owned contract.

list-indexes

list-indexes is always available and takes no arguments. Call it first on a multi-index server to discover which logical ids exist and how to filter each one.

Example response payload:

```
{
  "indexes": [
    {
      "id": "knowledge",
      "description": "Internal runbooks and operational guidance.",
      "upsert_available": true,
      "fields": [
        { "name": "title", "type": "text" },
        { "name": "category", "type": "tag" },
        { "name": "rating", "type": "numeric" }
      ],
      "limits": { "max_limit": 25 }
    },
    {
      "id": "tickets",
      "description": "Read-only mirror of resolved support tickets.",
      "upsert_available": false,
      "fields": [
        { "name": "category", "type": "tag" }
      ],
      "limits": { "max_limit": 50 }
    }
  ]
}
```

Notes:

- upsert_available reflects the binding's effective write availability (global read-only **or** the per-index read_only flag)
- fields lists the filterable fields discovered from the index; the vector field and the configured embed-source text field are intentionally omitted
- limits includes only runtime limits that were explicitly configured (such as max_limit or max_upsert_records); defaults are not echoed
- the underlying Redis index name (redis_name) is never exposed
- description appears only when configured for that binding

search-records

Arguments:

- `index` (optional; required when multiple indexes are configured)
- `query`
- `limit`
- `offset`
- `filter`
- `return_fields`

Example request payload:

```
{
  "index": "knowledge",
  "query": "incident response runbook",
  "limit": 2,
  "offset": 0,
  "filter": {
    "and": [
      { "field": "category", "op": "eq", "value": "operations" },
      { "field": "rating", "op": "gte", "value": 4 }
    ]
  },
  "return_fields": ["title", "content", "category", "rating"]
}
```

Example response payload:

```
{
  "index": "knowledge",
  "search_type": "hybrid",
  "offset": 0,
  "limit": 2,
  "results": [
    {
      "id": "knowledge:runbook:eu-failover",
      "score": 0.82,
      "score_type": "hybrid_score",
      "record": {
        "title": "EU failover runbook",
        "content": "Restore traffic after a regional failover.",
        "category": "operations",
        "rating": 5
      }
    }
  ]
}
```

Notes:

- `index` selects the logical binding; omit it only on a single-index server. The resolved id is echoed back in the response

- `search_type` is response metadata, not a request argument
- when `return_fields` is omitted, RedisVL MCP returns all non-vector fields
- returning the configured vector field is rejected
- `filter` accepts either a raw string or a JSON DSL object
- the `search-records` tool description includes schema-derived hints for typed JSON DSL filter fields, object-filter exists support, and valid `return_fields`
- `offset + limit` must stay within `runtime.max_result_window`
- startup rejects schemas that use MCP-reserved score metadata field names: `id`, `__key`, `key`, `score`, `vector_distance`, `__score`, `text_score`, `vector_similarity`, `hybrid_score`

upsert-records

Arguments:

- `index` (optional; required when multiple indexes are configured)
- `records`
- `id_field`
- `skip_embedding_if_present`

Example request payload:

```
{
  "index": "knowledge",
  "records": [
    {
      "doc_id": "doc-42",
      "content": "Updated operational guidance for failover handling.",
      "category": "operations",
      "rating": 5
    }
  ],
  "id_field": "doc_id"
}
```

Example response payload:

```
{
  "index": "knowledge",
  "status": "success",
  "keys_upserted": 1,
  "keys": ["knowledge:doc-42"]
}
```

Notes:

- `index` selects the logical binding; omit it only on a single-index server. The resolved id is echoed back in the response
- this tool is not registered when every binding is read-only (global read-only mode or every binding setting `read_only: true`)

- a write targeting a read-only binding is rejected with `forbidden` before any data is changed, even when the tool is registered because other bindings are writable
- when server-side embedding is configured, records that need embedding must contain `runtime.default_embed_text_field`
- when `skip_embedding_if_present` is `true`, records that already contain the configured vector field can skip re-embedding
- when a vector field is configured but server-side embedding is disabled, callers must supply vectors explicitly

Search Examples

Discovery-First Multi-Index Flow

On a multi-index server, call `list-indexes` first, pick a logical id from the response, then pass it as `index`:

```
{
  "index": "knowledge",
  "query": "cache invalidation incident",
  "limit": 3,
  "return_fields": ["title", "content", "category"]
}
```

The same `index` argument routes an `upsert-records` write to a specific binding:

```
{
  "index": "knowledge",
  "records": [
    { "doc_id": "doc-7", "content": "New runbook entry", "category": "operations" }
  ],
  "id_field": "doc_id"
}
```

On a single-index server you can omit `index` entirely; the examples below show that backward-compatible shape.

Read-Only Vector Search

Use read-only mode when assistants should only retrieve data:

```
uvx --from redisvl[mcp] rvl mcp --config /path/to/mcp.yaml --read-only
```

With a `search.type` of `vector`, callers send only the query, filters, pagination, and field projection:

```
{
  "query": "cache invalidation incident",
  "limit": 3,
  "return_fields": ["title", "content", "category"]
}
```

Raw String Filter

Pass a raw Redis filter string through unchanged:

```
{
  "query": "science",
  "filter": "@category:{science}",
  "return_fields": ["content", "category"]
}
```

JSON DSL Filter

The DSL supports logical operators and type-checked field operators:

```
{
  "query": "science",
  "filter": {
    "and": [
      { "field": "category", "op": "eq", "value": "science" },
      { "field": "rating", "op": "gte", "value": 4 }
    ]
  },
  "return_fields": ["content", "category", "rating"]
}
```

Pagination and Field Projection

```
{
  "query": "science",
  "limit": 1,
  "offset": 1,
  "return_fields": ["content", "category"]
}
```

Hybrid Search With `schema_overrides`

Use `schema_overrides` when Redis inspection cannot recover complete vector attrs, then keep hybrid behavior in config:

```
schema_overrides:
  fields:
    - name: embedding
      type: vector
      attrs:
        algorithm: flat
        dims: 1536
        datatype: float32
        distance_metric: cosine
```

(continues on next page)

(continued from previous page)

```
search:
  type: hybrid
  params:
    text_scorer: BM25STD
    stopwords: english
    vector_search_method: KNN
    combination_method: LINEAR
    linear_text_weight: 0.3
```

The MCP caller still sends the same request shape:

```
{
  "query": "legacy cache invalidation flow",
  "filter": { "field": "category", "op": "eq", "value": "release-notes" },
  "return_fields": ["title", "content", "release_version"]
}
```

Upsert Examples

Auto-Embed New Records

If a record does not include the configured vector field, RedisVL MCP embeds runtime.default_embed_text_field and writes the result:

```
{
  "records": [
    {
      "content": "First upserted document",
      "category": "science",
      "rating": 5
    },
    {
      "content": "Second upserted document",
      "category": "health",
      "rating": 4
    }
  ]
}
```

Update Existing Records With id_field

```
{
  "records": [
    {
      "doc_id": "doc-1",
      "content": "Updated content",
      "category": "engineering",
      "rating": 5
    }
  ]
}
```

(continues on next page)

(continued from previous page)

```
],  
  "id_field": "doc_id"  
}
```

Control Re-Embedding With `skip_embedding_if_present`

```
{  
  "records": [  
    {  
      "doc_id": "doc-2",  
      "content": "Existing content",  
      "category": "science",  
      "rating": 4  
    }  
  ],  
  "id_field": "doc_id",  
  "skip_embedding_if_present": false  
}
```

Set `skip_embedding_if_present` to `false` when you want the server to regenerate embeddings during `upsert`. In most cases, the caller should omit the vector field and let the server manage embeddings from `runtime.default_embed_text_field`.

Plain Writes Without Embedding

For fulltext-only indexes, `upsert-records` can write records without any vectorizer or vector field configuration:

```
{  
  "records": [  
    {  
      "content": "Updated FAQ entry",  
      "category": "support",  
      "rating": 5  
    }  
  ]  
}
```

If you configure a vector field but omit server-side embedding support, the caller must send vectors in each record instead of relying on the server to generate them.

Troubleshooting

Missing MCP Dependencies

If `rvl mcp` reports missing optional dependencies, install the MCP extra:

```
pip install redisvl[mcp]
```

If the configured vectorizer needs a provider SDK, install that provider extra too. Fulltext-only configs can omit the vectorizer entirely.

Configured Redis Index Does Not Exist

The server only binds to an existing index. Create the index first, then point `indexes.<id>.redis_name` at that index name.

Missing Required Environment Variables

YAML values support `${VAR}` and `${VAR:-default}` substitution. Missing required variables fail startup before the server registers tools.

Vectorizer Dimension Mismatch

If the vectorizer dims do not match the configured vector field dims, startup fails. Make sure the embedding model and the effective vector field dimensions are aligned.

Hybrid Config Requires Native Runtime Support

Some hybrid params depend on native hybrid support in Redis and redis-py. If your environment does not support that path, remove native-only params such as `knn_ef_runtime` or upgrade Redis and redis-py.

Authenticate RedisVL MCP

This guide explains how the RedisVL MCP server authenticates clients on its HTTP transports and how it gates read vs write access. It also draws the boundary between what RedisVL enforces and what belongs in a gateway or policy layer.

Note

Authentication applies only to the HTTP transports (`streamable-http`, `sse`). The `stdio` transport is a local subprocess with no network surface and is never authenticated.

Important

Authentication is a separate concern from **transport security** (Host/Origin validation), which is always on for the HTTP transports and defends against DNS rebinding independently of auth. See [Transport Security](#). Both layers

apply together: `auth` decides *who* may call; the `Host/Origin` guard rejects requests whose claimed authority is not allowlisted.

What RedisVL Enforces

RedisVL validates a bearer **JWT** that an existing identity provider (IdP) issued. It does not run an OAuth authorization server and does not issue tokens.

On each request it checks:

- **Signature**, against a JWKS endpoint or a static public key.
- **Issuer** (`iss`), so only tokens from your IdP are accepted.
- **Audience** (`aud`), so a token minted for a different service cannot be replayed against this server (RFC 8707).
- **Expiration**: a present `exp` in the past is rejected, and tokens are required to carry `exp` and `iat` (configurable via `required_claims`), so a token with no expiration, which would never expire, is rejected.
- **Required scopes** to connect, and (optionally) a **read scope** to call `search-records` and a **write scope** to call `upsert-records`.

Important

This is **coarse** authorization: it decides whether a caller may connect and whether it may read or write. It does **not** map token claims to a Redis ACL user, a per-tenant index, or query filters. See [The Authorization Boundary](#).

OAuth: Which Part RedisVL Handles

In OAuth terms, RedisVL MCP is a **resource server**: it validates access tokens that your identity provider issued. It does not run an OAuth login flow and does not issue tokens.

Steps 1 and 2 (obtaining the token) are handled by your client and IdP. RedisVL only performs the validation in step 3. As a result:

- It works with any OAuth 2.0 / OIDC provider (for example Auth0, Okta, Azure AD / Entra, Cognito, Keycloak): point `jwks_uri` at the provider and validate its tokens.
- RedisVL does not broker interactive “log in with...” flows and does not mint tokens.

You would only need more than token validation when you want the server itself to drive an interactive browser login (an OAuth proxy), or to act as its own authorization server that issues tokens. Both are out of scope today. If interactive login is ever needed, a single generic `oauth-proxy` option can be added behind the `auth.type` switch. For enterprise and agent deployments where the caller already holds a token, JWT validation is sufficient.

Request Flow

Configure JWT Authentication

Add a `server.auth` block to your MCP config. Secrets can be injected with `${ENV}` substitution.

```
server:
  redis_url: ${REDIS_URL:-redis://localhost:6379}
  auth:
    type: jwt
    jwks_uri: ${MCP_JWKS_URI}           # or set public_key for a static key
    issuer: ${MCP_ISSUER}
    audience: api://redisvl-mcp
    required_scopes: [kb.read]          # required to connect
    required_claims: [exp, iat]        # claims every token must carry (default)
    read_scope: kb.search.read         # required for search-records
    write_scope: kb.search.write       # required for upsert-records

indexes:
  knowledge:
    redis_name: docs_index
  search:
    type: fulltext
  runtime:
    text_field_name: content
```

Every field is also settable through `REDISVL_MCP_AUTH_*` environment variables, which take precedence over the YAML block.

Choosing the Authorization Claim

Different identity providers carry authorization in different claims. The default JWT scope claim is `scp` (or `scope`). Some enterprise providers carry authorization in a **roles** claim instead, which does not appear in the standard scope set.

Set the claim that holds your authorization values so read and write gating reads the right place:

```
server:
  auth:
    type: jwt
    # ...
    authorization_claim: roles          # default: scp
    read_scope: kb.search.read
    write_scope: kb.search.write
```

A token like the following would then pass the read gate, because `kb.search.read` is present in `roles`:

```
{
  "iss": "https://your-idp.example/{tenant}/v2.0",
  "aud": "api://redisvl-mcp",
  "sub": "nitin",
  "roles": ["kb.search.read"],
```

(continues on next page)

(continued from previous page)

```
"scp": "kb.read"
}
```

The Authorization Boundary

RedisVL MCP authenticates the caller and gates read vs write. It does **not** translate token claims (such as a tenant id or role) into a specific Redis ACL user, a per-tenant index, or injected query filters. The server holds one Redis connection for one index, established at startup.

Fine-grained, per-tenant data isolation belongs in a **gateway or policy layer** in front of the MCP server, which validates the token, looks up a binding of claim to Redis identity, and injects credentials and filters.

Use RedisVL's JWT validation for authentication and coarse read/write authorization. Layer a gateway on top when you need per-tenant Redis ACL enforcement.

When such a gateway or reverse proxy terminates the connection and forwards a rewritten `Host` header, set `server.transport_security.enabled: false` (or declare the public host via `server.transport_security.allowed_hosts`) so the proxy's rewritten `Host` is not rejected by the `Host/Origin` guard.

See Also

- *Run RedisVL MCP*: run and configure the RedisVL MCP server.

Migrate an Index

Warning

The index migrator is an **experimental** feature. APIs, CLI commands, and on-disk formats (plans, checkpoints, backups) may change in future releases. Review migration plans carefully before applying to production indexes.

This guide shows how to safely change your index schema using the RedisVL migrator.

Quick Start

Add a field to your index in 4 commands:

```
# 1. See what indexes exist
rvl index listall --url redis://localhost:6379

# 2. Use the wizard to build a migration plan
rvl migrate wizard --index myindex --url redis://localhost:6379

# 3. Apply the migration
rvl migrate apply --plan migration_plan.yaml --backup-dir ./migration_backups --url
↪redis://localhost:6379

# 4. Verify the result
rvl migrate validate --plan migration_plan.yaml --url redis://localhost:6379
```

Prerequisites

- Redis with the Search module (Redis Stack, Redis Cloud, or Redis Enterprise)
- An existing index to migrate
- `redisvl` installed (`pip install redisvl`)

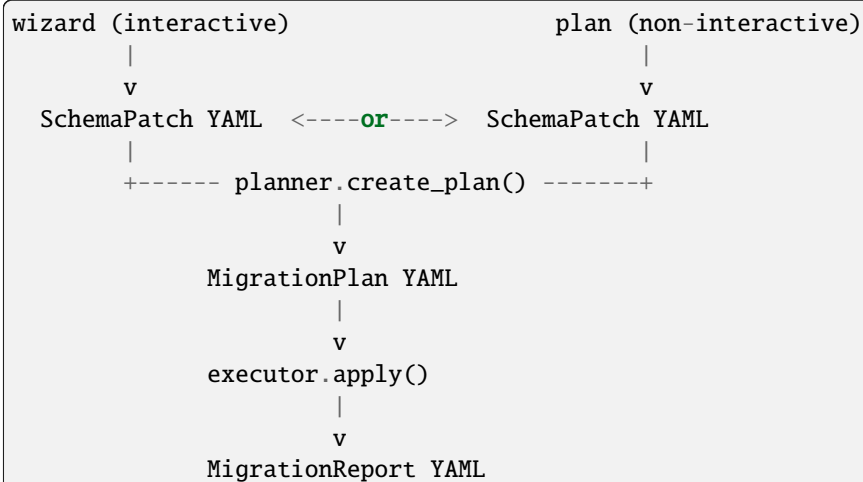
```
# Local development with Redis 8.0+ (recommended for full feature support)
docker run -d --name redis -p 6379:6379 redis:8.0
```

Note: Redis 8.0+ is required for INT8/UINT8 vector datatypes. SVS-VAMANA algorithm requires Redis 8.2+ and Intel AVX-512 hardware.

How It Works

Every migration follows the same three-phase flow: **describe what changed** (the patch), **generate a plan** (diffing the patch against the live schema), and **execute the plan**.

Single-Index Flow: wizard/plan then apply



Phase 1: Build a SchemaPatch. A patch is a small YAML file that declares *what you want to change*, not the full target schema. You can build it interactively with `rvl migrate wizard`, or write it by hand. The patch has five sections, each optional:

Patch Section	What it does
<code>add_fields</code>	Adds new field definitions to the index
<code>remove_fields</code>	Removes fields from the index (document data is kept, just no longer indexed)
<code>rename_fields</code>	Renames fields in both the index schema and all documents (HGET old, HSET new, HDEL old)
<code>update_fields</code>	Modifies field attributes: algorithm, datatype, distance metric, sortable, separator, etc.
<code>index</code>	Changes the index name or key prefix

Phase 2: Generate a MigrationPlan. The planner connects to Redis, snapshots the live index schema and stats, then merges the patch into the source schema to produce a `merged_target_schema`. It classifies every change as supported or blocked and extracts rename operations.

The plan YAML contains:

- **source**: frozen snapshot of the live index at planning time (schema, stats, key sample, prefixes)
- **requested_changes**: the patch that was applied
- **merged_target_schema**: source + patch = what the index will look like after migration
- **diff_classification**: whether the migration is supported and any blocked reasons
- **rename_operations**: extracted index renames, prefix changes, and field renames
- **warnings**: any important notes (downtime required, lossy quantization, etc.)

The same patch produces different plans per index because each index has a different source schema.

Phase 3: Apply. The executor reads the plan and runs the migration steps:

1. Enumerate keys (SCAN with source prefix)
2. Field renames (pipelined HGET/HSET/HDEL)
3. Prepare vector backups, if hash vector bytes will be quantized
4. Drop index (FT.DROPINDEX, documents are preserved)
5. Key prefix renames (RENAME or DUMP/RESTORE for cluster)
6. Quantize vectors from backup (pipelined read/convert/write)
7. Create index (FT.CREATE with merged target schema)
8. Wait for re-indexing to complete
9. Validate (doc count, schema match, key sample)

--backup-dir / backup_dir is required before any apply starts. For quantization, the directory stores original vector bytes for resume and rollback. For index-only migrations, the directory is still validated and recorded in the report, but no vector backup files are written.

Warning

Hash vector quantization is supported only when the Redis keys being quantized are not also indexed by another live RediSearch index that expects the old vector datatype. Quantization rewrites vector bytes in the document itself; any other index that covers the same key sees those new bytes and may silently drop the document or fail to index it. If the same documents are intentionally shared across multiple indexes, do not use the migrator for that quantization change. Use an application-level migration that creates new keys or fields and coordinates every affected index schema.

Batch Flow: wizard/plan then batch-plan then batch-apply

For applying the same change across multiple indexes:

```
SchemaPatch YAML  (shared, written once)
|
v
batch_planner.create_batch_plan()
for each index:
  snapshot live schema
  merge patch into source
```

(continues on next page)

(continued from previous page)

```

    if applicable: write per-index MigrationPlan
    if not: mark skip_reason
        |
        v
BatchPlan YAML
shared_patch: { ... }
indexes:
  - name: idx_a, applicable: true, plan_path: plans/idx_a.yaml
  - name: idx_b, applicable: true, plan_path: plans/idx_b.yaml
  - name: idx_c, applicable: false, skip_reason: "field not found"
    |
    v
batch_executor.apply()
for each applicable index (sequentially):
    executor.apply(per_index_plan)

```

The batch planner takes a **single shared patch** and tests it against every target index. Indexes where the patch doesn't apply (e.g., it references a field that doesn't exist in that index, or the change is blocked) are marked **applicable: false** with a **skip_reason** and skipped during apply. Each applicable index gets its own full **MigrationPlan** written to disk.

This means you can review each per-index plan individually before running **batch-apply**.

Step 1: Discover Available Indexes

```
rvl index listall --url redis://localhost:6379
```

Example output:

```

Indices:
  1. products_idx
  2. users_idx
  3. orders_idx

```

Step 2: Build Your Schema Change

Choose one of these approaches:

Option A: Use the Wizard (Recommended)

The wizard guides you through building a migration interactively. Run:

```
rvl migrate wizard --index myindex --url redis://localhost:6379
```

Example wizard session (adding a field):

```

Building a migration plan for index 'myindex'
Current schema:
- Index name: myindex

```

(continues on next page)

(continued from previous page)

```
- Storage type: hash
  - title (text)
  - embedding (vector)

Choose an action:
1. Add field      (text, tag, numeric, geo)
2. Update field   (sortable, weight, separator, vector config)
3. Remove field
4. Rename field   (rename field in all documents)
5. Rename index   (change index name)
6. Change prefix  (rename all keys)
7. Preview patch  (show pending changes as YAML)
8. Finish
Enter a number: 1

Field name: category
Field type options: text, tag, numeric, geo
Field type: tag
  Sortable: enables sorting and aggregation on this field
Sortable [y/n]: n
  Separator: character that splits multiple values (default: comma)
Separator [leave blank to keep existing/default]: |

Choose an action:
1. Add field      (text, tag, numeric, geo)
2. Update field   (sortable, weight, separator, vector config)
3. Remove field
4. Rename field   (rename field in all documents)
5. Rename index   (change index name)
6. Change prefix  (rename all keys)
7. Preview patch  (show pending changes as YAML)
8. Finish
Enter a number: 8

Migration plan written to /path/to/migration_plan.yaml
Mode: drop_recreate
Supported: True
Warnings:
- Index downtime is required
```

Example wizard session (quantizing vectors):

```
Choose an action:
1. Add field      (text, tag, numeric, geo)
2. Update field   (sortable, weight, separator, vector config)
3. Remove field
4. Rename field   (rename field in all documents)
5. Rename index   (change index name)
6. Change prefix  (rename all keys)
7. Preview patch  (show pending changes as YAML)
8. Finish
Enter a number: 2
```

(continues on next page)

(continued from previous page)

```

Updatable fields:
1. title (text)
2. embedding (vector)
Select a field to update by number or name: 2

Current vector config for 'embedding':
  algorithm: HNSW
  datatype: float32
  distance_metric: cosine
  dims: 384 (cannot be changed)
  m: 16
  ef_construction: 200

Leave blank to keep current value.
  Algorithm: vector search method (FLAT=brute force, HNSW=graph, SVS-VAMANA=compressed_
↪graph)
Algorithm [current: HNSW]:
  Datatype: float16, float32, bfloat16, float64, int8, uint8
            (float16 reduces memory ~50%, int8/uint8 reduce ~75%)
Datatype [current: float32]: float16
  Distance metric: how similarity is measured (cosine, l2, ip)
Distance metric [current: cosine]:
  M: number of connections per node (higher=better recall, more memory)
M [current: 16]:
  EF_CONSTRUCTION: build-time search depth (higher=better recall, slower build)
EF_CONSTRUCTION [current: 200]:

Choose an action:
...
8. Finish
Enter a number: 8

Migration plan written to /path/to/migration_plan.yaml
Mode: drop_recreate
Supported: True

```

Option B: Write a Schema Patch (YAML)

Create `schema_patch.yaml` manually:

```

version: 1
changes:
  add_fields:
    - name: category
      type: tag
      path: $.category
      attrs:
        separator: "|"
  remove_fields:
    - legacy_field

```

(continues on next page)

(continued from previous page)

```

update_fields:
  - name: title
    attrs:
      sortable: true
  - name: embedding
    attrs:
      datatype: float16      # quantize vectors
      algorithm: HNSW
      distance_metric: cosine

```

Then generate the plan:

```

rvl migrate plan \
  --index myindex \
  --schema-patch schema_patch.yaml \
  --url redis://localhost:6379 \
  --plan-out migration_plan.yaml

```

Option C: Provide a Target Schema

If you have the complete target schema, use it directly:

```

rvl migrate plan \
  --index myindex \
  --target-schema target_schema.yaml \
  --url redis://localhost:6379 \
  --plan-out migration_plan.yaml

```

Step 3: Review the Migration Plan

Before applying, review migration_plan.yaml:

```

# migration_plan.yaml (example)
version: 1
mode: drop_recreate

source:
  schema_snapshot:
    index:
      name: myindex
      prefix: "doc:"
      storage_type: json
    fields:
      - name: title
        type: text
      - name: embedding
        type: vector
        attrs:
          dims: 384
          algorithm: hnsw

```

(continues on next page)

(continued from previous page)

```

        datatype: float32
stats_snapshot:
  num_docs: 10000
keyspace:
  prefixes: ["doc:"]
  key_sample: ["doc:1", "doc:2", "doc:3"]

requested_changes:
  add_fields:
    - name: category
      type: tag

diff_classification:
  supported: true
  blocked_reasons: []

rename_operations:
  rename_index: null
  change_prefix: null
  rename_fields: []

merged_target_schema:
  index:
    name: myindex
    prefix: "doc:"
    storage_type: json
  fields:
    - name: title
      type: text
    - name: category
      type: tag
    - name: embedding
      type: vector
      attrs:
        dims: 384
        algorithm: hnsw
        datatype: float32

warnings:
  - "Index downtime is required"

```

Key fields to check:

- `diff_classification.supported` - Must be true to proceed
- `diff_classification.blocked_reasons` - Must be empty
- `warnings` - Top-level warnings about the migration
- `merged_target_schema` - The final schema after migration

Understanding Downtime Requirements

CRITICAL: During a `drop_recreate` migration, your application must:

Requirement	Description
Pause reads	Index is unavailable during migration
Pause writes	Writes during migration may be missed or cause conflicts

Why Both Reads AND Writes Must Be Paused

- **Reads:** The index definition is dropped and recreated. Any queries during this window will fail.
- **Writes:** Redis updates indexes synchronously on every write. If your app writes documents while the index is dropped, those writes are not indexed. Additionally, if you're quantizing vectors (float32 → float16), concurrent writes may conflict with the migration's re-encoding process.

What “Downtime” Means

Downtime Type	Reads	Writes	Safe?
Full quiesce (recommended)	Stopped	Stopped	YES
Read-only pause	Stopped	Continuing	NO
Active	Active	Active	NO

Recovery from Interrupted Migration

Interruption Point	Documents	Index	Recovery
After drop, before quantize	Unchanged	None	Re-run apply with the same <code>--backup-dir</code>
During quantization	Partially quantized	None	Re-run with same <code>--backup-dir</code> to resume from last batch
After quantization, before create	Quantized	None	Re-run apply (will recreate index)
After create	Correct	Rebuilding	Wait for index ready

The underlying documents are **never deleted** by `drop_recreate` mode. `--backup-dir` is required for apply and enables crash-safe recovery for vector quantization. See [Crash-safe resume for quantization](#) below.

Step 4: Apply the Migration

The `apply` command executes the migration. The index will be temporarily unavailable during the drop-recreate process.

```
rvl migrate apply \
--plan migration_plan.yaml \
--url redis://localhost:6379 \
--backup-dir ./migration_backups \
--report-out migration_report.yaml \
--benchmark-out benchmark_report.yaml
```

What apply does

The migration executor follows this sequence:

STEP 1: Enumerate keys (before any modifications)

- Discovers all document keys belonging to the source index
- Uses `FT.AGGREGATE WITHCURSOR` for efficient enumeration
- Falls back to `SCAN` if the index has indexing failures
- Keys are stored in memory for quantization or rename operations

STEP 2: Field renames (if renaming fields)

- Renames document fields before the source index is dropped
- Uses pipelined `HGET/HSET/HDEL` for Hash storage or JSON path updates for JSON storage
- Skipped if the plan has no field rename operations

STEP 3: Back up original vectors (if hash vector bytes will be quantized)

- Single-worker hash quantization writes original vector bytes to `<backup-dir>` before the index is dropped
- Multi-worker hash quantization writes per-worker backup shards during the quantization phase after the drop
- JSON datatype changes and index-only migrations validate and record `--backup-dir` but do not write vector backup files

STEP 4: Drop source index

- Issues `FT.DROPINDEX` to remove the index structure
- **The underlying documents remain in Redis** - only the index metadata is deleted
- After this point, the index is unavailable until the target index is recreated and ready

STEP 5: Key renames (if changing key prefix)

- If the migration changes the key prefix, renames each key from old prefix to new prefix
- Skipped if no prefix change

STEP 6: Quantize vectors (if changing hash vector datatype)

- For each document in the enumerated key list:
 - Reads the document (including the old vector)
 - Converts the vector to the new datatype (e.g., `float32` → `float16`)

- Writes back the converted vector to the same document
- Processes documents in batches of 500 using Redis pipelines
- Skipped for JSON storage (vectors are re-indexed automatically on recreate)
- **Backup support:** `--backup-dir` is required and enables crash-safe recovery and rollback for vector quantization
- **Shared-key limitation:** unsupported if the same Redis keys are also indexed by another live index that expects the old vector datatype

STEP 7: Create target index

- Issues `FT.CREATE` with the merged target schema
- Redis begins background indexing of existing documents

STEP 8: Wait for re-indexing

- Polls `FT.INFO` until indexing completes
- The index becomes available for queries when this completes

Summary: The migration preserves all documents, drops only the index structure, performs any document-level transformations (quantization, renames), then recreates the index with the new schema.

Async execution for large migrations

For large migrations (especially those involving vector quantization), use the `--async` flag:

```
rvl migrate apply \  
  --plan migration_plan.yaml \  
  --async \  
  --backup-dir ./migration_backups \  
  --url redis://localhost:6379
```

What becomes async:

- Document enumeration during quantization (uses `FT.AGGREGATE WITHCURSOR` for index-specific enumeration, falling back to `SCAN` only if indexing failures exist)
- Vector read/write operations (sequential async `HGET`, batched `HSET` via pipeline)
- Index readiness polling (uses `asyncio.sleep()` instead of blocking)
- Validation checks

What stays sync:

- CLI prompts and user interaction
- YAML file reading/writing
- Progress display

When to use async:

- Quantizing millions of vectors (float32 to float16)
- Integrating into an async application

For most migrations (index-only changes, small datasets), sync mode is sufficient and simpler.

See [Index Migrations](#) for detailed async vs sync guidance.

Crash-safe resume for quantization

When migrating large datasets with vector quantization (e.g. float32 to float16), the re-encoding step can take minutes or hours. If the process is interrupted (crash, network drop, OOM kill), you don't want to start over. The `--backup-dir` flag enables crash-safe recovery.

How it works

For hash vector datatype changes, the migrator saves original vector bytes to disk before mutating them. Single-worker migrations create two files:

```
<backup-dir>/
migration_backup_<index_name>.header  # JSON: phase, progress counters, field metadata
migration_backup_<index_name>.data    # Binary: length-prefixed batches of original
↳ vectors
```

Multi-worker migrations also create a `.manifest` file at the canonical backup path. The manifest records worker shard paths and key slices so a retry can resume even if the source index was already dropped.

The **header file** is a small JSON file that tracks progress through a state machine:

```
dump → ready → index_dropped → active → completed → target_created → validated
```

- **dump**: original vectors are being read from Redis and written to the data file, one batch at a time
- **ready**: all original vectors have been backed up; the source index may still be live
- **index_dropped**: the source index definition has been dropped, but vectors have not all been rewritten
- **active**: quantization is in progress; the header tracks which batches have been written back to Redis
- **completed**: all batches have been quantized; target index creation may still be pending
- **target_created**: the target index was recreated and Redis is re-indexing or ready for validation
- **validated**: post-migration validation passed

The header is atomically updated (temp file + rename) after every batch, so a crash never corrupts it.

The **data file** is append-only binary. Each batch is stored as a 4-byte big-endian length prefix followed by a pickled blob containing the batch's keys and their original vector bytes.

On resume, the executor loads the header, sees how many batches were already quantized (`quantize_completed_batches`), and skips ahead in the data file to continue from the next unfinished batch.

Disk usage: approximately $\text{num_docs} \times \text{dims} \times \text{bytes_per_element}$. For example, 1M docs with 768-dim float32 vectors 2.9 GB.

Step-by-step: using crash-safe resume

1. Estimate disk space (dry-run, no mutations):

```
rvl migrate estimate --plan migration_plan.yaml
```

Example output:

Pre-migration disk space estimate:

Index: products_idx (1,000,000 documents)

Vector field 'embedding': 768 dims, float32 -> float16

RDB snapshot (BGSAVE): ~2.87 GB

AOF growth: not estimated (pass aof_enabled=True if AOF is on)

Total new disk required: ~2.87 GB

Post-migration memory savings: ~1.43 GB (50% reduction)

If AOF is enabled:

```
rvl migrate estimate --plan migration_plan.yaml --aof-enabled
```

2. Apply with backup enabled:

```
rvl migrate apply \  
--plan migration_plan.yaml \  
--backup-dir /tmp/migration_backups \  
--url redis://localhost:6379 \  
--report-out migration_report.yaml
```

The `--backup-dir` flag takes a directory path. If no backup exists there, a new one is created. If one already exists (from a previous interrupted run), the migrator resumes from where it left off. A completed backup is treated as a no-op resume only when the live index already matches the target schema; after rollback, the live index matches the source schema, so the old completed backup is treated as stale and a fresh backup is written.

3. If the process crashes or is interrupted:

The header file will contain the progress:

```
{  
  "index_name": "products_idx",  
  "fields": {"embedding": {"source": "float32", "target": "float16", "dims": 768}},  
  "batch_size": 500,  
  "phase": "active",  
  "dump_completed_batches": 2000,  
  "quantize_completed_batches": 900  
}
```

This tells you: all 2000 batches of original vectors were backed up, and 900 of them have been quantized so far.

4. Resume the migration:

Re-run the exact same command:

```
rvl migrate apply \  
--plan migration_plan.yaml \  
--backup-dir /tmp/migration_backups \  
--url redis://localhost:6379 \  
--report-out migration_report.yaml
```

The migrator will:

- Detect the existing backup and skip already-quantized batches
- Continue quantizing from batch 901 onward

- Print progress like `Quantize vectors: 450,000/1,000,000 docs`

5. On successful completion:

The backup phase is set to **completed**. Backup files are **always retained** on disk for post-migration auditing and rollback. Delete them manually from `--backup-dir` once you have verified the migrated data and no longer need a recovery path.

Limitations

- **Same-width conversions** (float16 to bfloat16, or int8 to uint8) are **not supported** for resume. These conversions cannot be detected by byte-width inspection, so idempotent skip is impossible.
- **Shared keys across indexes** are **not supported** for hash vector quantization. The migrator mutates vector bytes in the Redis document key; if another index also covers that key and still expects the old datatype, the document may be dropped from that index or fail to re-index.
- **JSON storage** does not need vector re-encoding (Redis re-indexes JSON vectors on `FT.CREATE`). The backup directory is still required, validated, and recorded, but no vector backup files are written.
- The backup must match the migration plan. If you change the plan, delete the old backup directory and start fresh.

Step 5: Validate the Result

Validation happens automatically during `apply`, but you can run it separately:

```
rvl migrate validate \
  --plan migration_plan.yaml \
  --url redis://localhost:6379 \
  --report-out migration_report.yaml
```

Validation checks:

- Live schema matches `merged_target_schema`
- Document count matches the source snapshot
- Sampled keys still exist
- No increase in indexing failures

What's Supported

Change	Supported	Notes
Add text/tag/numeric/geo field		
Remove a field		
Rename a field		Renames field in all documents
Change key prefix		Renames keys via RENAME command
Rename the index		Index-only
Make a field sortable		
Change field options (separator, stemming)		
Change vector algorithm (FLAT HNSW SVS-VAMANA)		Index-only
Change distance metric (COSINE L2 IP)		Index-only
Tune HNSW parameters (M, EF_CONSTRUCTION)		Index-only
Quantize vectors (float32 → float16/bfloat16/int8/uint8)		Auto re-encode; unsupported when the same Redis keys are indexed by another live index expecting the old datatype

What's Blocked

Change	Why	Workaround
Change vector dimensions	Requires re-embedding	Re-embed with new model, reload data
Change storage type (hash JSON)	Different data format	Export, transform, reload
Add a new vector field	Requires vectors for all docs	Add vectors first, then migrate

CLI Reference

Single-Index Commands

Command	Description
<code>rvl migrate wizard</code>	Build a migration interactively
<code>rvl migrate plan</code>	Generate a migration plan
<code>rvl migrate apply</code>	Execute a migration
<code>rvl migrate estimate</code>	Estimate disk space for a migration (dry-run)
<code>rvl migrate validate</code>	Verify a migration result

Batch Commands

Command	Description
<code>rvl migrate batch-plan</code>	Create a batch migration plan
<code>rvl migrate batch-apply</code>	Execute a batch migration
<code>rvl migrate batch-resume</code>	Resume an interrupted batch
<code>rvl migrate batch-status</code>	Check batch progress

Common flags:

- `--url` : Redis connection URL
- `--index` : Index name to migrate
- `--plan` / `--plan-out` : Path to migration plan
- `--async` : Use async executor for large migrations (apply only)
- `--report-out` : Path for validation report
- `--benchmark-out` : Path for performance metrics

Apply flags (quantization & reliability):

- `--backup-dir <dir>` : Required migration backup directory. Hash vector datatype changes write vector backup files there for resume and rollback; index-only and JSON migrations validate and record the directory without writing vector backup files.
- `--batch-size <N>` : Keys per pipeline batch (default 500). Values 200 to 1000 are typical.
- `--workers <N>` : Parallel quantization workers (default 1). Each worker opens its own Redis connection. See [Performance](#) for guidance.

Batch-specific flags:

- `--pattern` : Glob pattern to match index names (e.g., `*_idx`)
- `--indexes` : Explicit list of index names
- `--indexes-file` : File containing index names (one per line)
- `--schema-patch` : Path to shared schema patch YAML
- `--state` : Path to batch state file for resume
- `--failure-policy` : `fail_fast` or `continue_on_error`
- `--accept-data-loss` : Required for quantization (lossy changes)
- `--retry-failed` : Retry previously failed indexes on resume

Troubleshooting

Migration blocked: “unsupported change”

The planner detected a change that requires data transformation. Check `diff_classification.blocked_reasons` in the plan for details.

Apply failed: “source schema mismatch”

The live index schema changed since the plan was generated. Re-run `rvl migrate plan` to create a fresh plan.

Apply failed: “timeout waiting for index ready”

The index is taking longer to rebuild than expected. This can happen with large datasets. Check Redis logs and consider increasing the timeout or running during lower traffic periods.

Validation failed: “document count mismatch”

Documents were added or removed between plan and apply. This is expected if your application is actively writing. Re-run `plan` and `apply` during a quieter period when the document count is stable, or verify the mismatch is due only to normal application traffic.

Quantized documents disappeared from another index

This topology is unsupported. Hash vector quantization rewrites vector bytes in the Redis document key. If another live RediSearch index also covers that key and still expects the old vector datatype, Redis may drop that document from the other index or report indexing failures for it.

Recover by rolling back the vector bytes from the migration backup, then recreate any affected index schemas. To perform the change safely, use an application-level migration that writes new physical keys or new vector fields and coordinates all affected indexes before switching traffic.

batch-plan failed: “overlapping indexes detected”

`batch-plan` refuses to write a plan when two or more applicable indexes share a key prefix (one prefix is a literal string-prefix of the other, matching `FT.CREATE PREFIX` semantics). Running such a batch would double-quantize the shared keys and corrupt vector data. The error lists each conflicting index pair under a `Conflicts:` section:

```
Error: Refusing to create batch plan: overlapping indexes detected.
```

```
Multiple indexes in the batch share Redis key prefixes. Running a
batch migration over overlapping indexes can mutate the same keys
more than once (e.g., double-quantization of vectors), corrupting
the underlying data.
```

```
Conflicts:
```

```
- products_main <-> products_premium: 'product:' <-> 'product:premium:'
```

(continues on next page)

(continued from previous page)

Resolve by migrating overlapping indexes one at a time, **or** by narrowing the batch to a **set** of indexes **with** disjoint prefixes.

Split the selected indexes into prefix-disjoint groups (for example, `prod_*` separately from `staging_*`) and run `batch-plan` once per group. Indexes that are skipped for other reasons (e.g. `applicable: false` because a field is missing) do not participate in this check.

How to recover from a failed migration

If `apply` fails mid-migration:

1. **Check if the index exists:** `rvl index info --index myindex`
2. **If the index exists but is wrong:** Re-run `apply` with the same plan
3. **If the index was dropped:** Recreate it from the plan's `merged_target_schema`

The underlying documents are never deleted by `drop_recreate`.

Backup, Resume & Rollback

How Backups Work

`--backup-dir / backup_dir` is required for all migrations. If it is omitted or empty, the executor raises `ValueError` before any migration starts. Migration reports include the resolved backup directory and backup file prefixes. Batch checkpoint state also stores the backup directory used by the run, and resume refuses a different directory for the same checkpoint.

For hash vector datatype changes, the migration executor saves **original vector bytes** to disk before mutating them. This enables two key capabilities:

1. **Crash-safe resume:** if the process dies mid-migration, re-running the same command with the same `--backup-dir` automatically detects partial progress and resumes from the last completed batch.
2. **Manual rollback:** the backup files contain the original (pre-quantization) vector values, which can be restored to undo a migration.

For index-only migrations and JSON datatype changes, the directory is still validated and recorded, but no `.header` or `.data` vector backup files are written.

Backup files are written to the specified directory with this layout:

```
<backup-dir>/
migration_backup_<index_name>.header    # JSON: phase, progress counters, field metadata
migration_backup_<index_name>.data      # Binary: length-prefixed batches of original
↳ vectors
migration_backup_<index_name>.manifest # JSON: multi-worker shard resume metadata,
↳ when workers > 1
```

Disk usage: approximately $\text{num_docs} \times \text{dims} \times \text{bytes_per_element}$. For example, 1M docs with 768-dim float32 vectors 2.9 GB.

Backup files are **always retained** on disk after a successful migration so they remain available for post-migration auditing and rollback. Delete the files manually from the backup directory once you no longer need a recovery path.

Crash-Safe Resume

If a migration is interrupted (crash, network error, Ctrl+C), simply re-run the exact same command:

```
# Original command that was interrupted
rvl migrate apply --plan plan.yaml --url redis://localhost:6379 \
  --backup-dir /tmp/backups --workers 4

# Just re-run it. Progress is resumed automatically
rvl migrate apply --plan plan.yaml --url redis://localhost:6379 \
  --backup-dir /tmp/backups --workers 4
```

The executor detects the existing backup header, reads how many batches were completed, and resumes from the next unfinished batch. No data is duplicated or lost. If a retained completed backup is found after rollback, the executor does not skip the migration unless the live index already matches the target schema; it treats the completed backup as stale and starts a fresh backup.

Note

Single-worker vs multi-worker resume: In single-worker mode, the full backup is written *before* the index is dropped, so a crash at any point leaves a complete backup on disk. In multi-worker mode, dump and quantize are fused (each worker reads, backs up, and converts its shard in one pass *after* the index drop). A crash during this fused phase may leave partial backup shards. Re-running detects and resumes from partial state.

Rollback

If you need to undo a quantization migration and restore original vectors, use the `rollback` command:

```
rvl migrate rollback --backup-dir /tmp/backups --url redis://localhost:6379
```

This reads every batch from the backup files and pipeline-HSETs the original (pre-quantization) vector bytes back into Redis. After rollback completes:

- Your vector data is restored to its original datatype
- You will need to **manually recreate the original index schema** if the index was changed during migration (the rollback command restores data only, not the index definition)

```
# After rollback, recreate the original index if needed:
rvl index create --schema original_schema.yaml --url redis://localhost:6379
```

Important

Rollback requires that the backup directory still contains the original backup files. Backups are retained automatically after migration; do not delete the directory until you are certain rollback is no longer needed.

Python API for Rollback

```
from redisvl.migration.backup import VectorBackup
import redis

r = redis.from_url("redis://localhost:6379")
backup = VectorBackup.load("/tmp/backups/migration_backup_myindex")

for keys, originals in backup.iter_batches():
    pipe = r.pipeline(transaction=False)
    for key in keys:
        if key in originals:
            for field_name, original_bytes in originals[key].items():
                pipe.hset(key, field_name, original_bytes)
    pipe.execute()

print("Rollback complete")
```

Python API

For programmatic migrations, use the migration classes directly:

Sync API

```
from redisvl.migration import MigrationPlanner, MigrationExecutor

planner = MigrationPlanner()
plan = planner.create_plan(
    "myindex",
    redis_url="redis://localhost:6379",
    schema_patch_path="schema_patch.yaml",
)

executor = MigrationExecutor()
report = executor.apply(
    plan,
    redis_url="redis://localhost:6379",
    backup_dir="/tmp/migration_backups",
)
print(f"Migration result: {report.result}")
```

With backup and multi-worker quantization:

```
report = executor.apply(
    plan,
    redis_url="redis://localhost:6379",
    backup_dir="/tmp/migration_backups",  # enables crash-safe resume
    batch_size=500,                      # keys per pipeline batch
    num_workers=4,                       # parallel quantization workers
```

(continues on next page)

(continued from previous page)

```
)
print(f"Quantized in {report.timings.quantize_duration_seconds}s")
```

Async API

```
import asyncio
from redisvl.migration import AsyncMigrationPlanner, AsyncMigrationExecutor

async def migrate():
    planner = AsyncMigrationPlanner()
    plan = await planner.create_plan(
        "myindex",
        redis_url="redis://localhost:6379",
        schema_patch_path="schema_patch.yaml",
    )

    executor = AsyncMigrationExecutor()
    report = await executor.apply(
        plan,
        redis_url="redis://localhost:6379",
        backup_dir="/tmp/migration_backups",
        num_workers=4,
    )
    print(f"Migration result: {report.result}")

asyncio.run(migrate())
```

Batch Migration

When you need to apply the same schema change to multiple indexes, use batch migration. This is common for:

- Quantizing all indexes from float32 → float16
- Standardizing vector algorithms across indexes
- Coordinated migrations during maintenance windows

Quick Start: Batch Migration

```
# 1. Create a shared patch (applies to any index with an 'embedding' field)
cat > quantize_patch.yaml << 'EOF'
version: 1
changes:
  update_fields:
    - name: embedding
      attrs:
        datatype: float16
EOF
```

(continues on next page)

(continued from previous page)

```
# 2. Create a batch plan for all indexes matching a pattern
rvl migrate batch-plan \
  --pattern "*_idx" \
  --schema-patch quantize_patch.yaml \
  --plan-out batch_plan.yaml \
  --url redis://localhost:6379

# 3. Apply the batch plan
rvl migrate batch-apply \
  --plan batch_plan.yaml \
  --backup-dir ./migration_backups \
  --accept-data-loss \
  --url redis://localhost:6379

# 4. Check status
rvl migrate batch-status --state batch_state.yaml
```

Batch Plan Options

Select indexes by pattern:

```
rvl migrate batch-plan \
  --pattern "*_idx" \
  --schema-patch quantize_patch.yaml \
  --plan-out batch_plan.yaml \
  --url redis://localhost:6379
```

Select indexes by explicit list:

```
rvl migrate batch-plan \
  --indexes "products_idx,users_idx,orders_idx" \
  --schema-patch quantize_patch.yaml \
  --plan-out batch_plan.yaml \
  --url redis://localhost:6379
```

Select indexes from a file (for 100+ indexes):

```
# Create index list file
echo -e "products_idx\nusers_idx\norders_idx" > indexes.txt

rvl migrate batch-plan \
  --indexes-file indexes.txt \
  --schema-patch quantize_patch.yaml \
  --plan-out batch_plan.yaml \
  --url redis://localhost:6379
```

Batch Plan Review

The generated `batch_plan.yaml` shows which indexes will be migrated:

```
version: 1
batch_id: "batch_20260320_100000"
mode: drop_recreate
failure_policy: fail_fast
requires_quantization: true

shared_patch:
  version: 1
  changes:
    update_fields:
      - name: embedding
        attrs:
          datatype: float16

indexes:
  - name: products_idx
    applicable: true
    skip_reason: null
  - name: users_idx
    applicable: true
    skip_reason: null
  - name: legacy_idx
    applicable: false
    skip_reason: "Field 'embedding' not found"

created_at: "2026-03-20T10:00:00Z"
```

Key fields:

- `applicable: true` means the patch applies to this index
- `skip_reason` explains why an index will be skipped

Overlap check. `batch-plan` refuses to write a plan when two applicable indexes have key prefixes that overlap — i.e. one prefix is a literal string-prefix of the other, matching `FT.CREATE PREFIX` semantics. Migrating overlapping indexes in a single batch can corrupt vector data because every index after the first reads bytes that an earlier index has already quantized. Split the indexes into prefix-disjoint groups and create a batch plan per group. See the troubleshooting entry below for the exact error message.

Applying a Batch Plan

```
# Apply with fail-fast (default: stop on first error)
rvl migrate batch-apply \
  --plan batch_plan.yaml \
  --backup-dir ./migration_backups \
  --accept-data-loss \
  --url redis://localhost:6379

# Apply with continue-on-error (set at batch-plan time)
```

(continues on next page)

(continued from previous page)

```
# Note: failure_policy is set during batch-plan, not batch-apply
rvl migrate batch-plan \
  --pattern "*_idx" \
  --schema-patch quantize_patch.yaml \
  --failure-policy continue_on_error \
  --plan-out batch_plan.yaml \
  --url redis://localhost:6379

rvl migrate batch-apply \
  --plan batch_plan.yaml \
  --backup-dir ./migration_backups \
  --accept-data-loss \
  --url redis://localhost:6379
```

Flags for batch-apply:

- `--accept-data-loss` : Required when quantizing vectors (float32 → float16 is lossy)
- `--backup-dir` : Required directory for per-index backup metadata and vector backup files when hash vector bytes are mutated
- `--state` : Path to batch state file (default: `batch_state.yaml`)
- `--report-dir` : Directory for per-index reports (default: `./reports/`)

Note: `--failure-policy` is set during `batch-plan`, not `batch-apply`. The policy is stored in the batch plan file.

Resume After Failure

Batch migration automatically tracks progress in the state file. If interrupted:

```
# Resume from where it left off
rvl migrate batch-resume \
  --state batch_state.yaml \
  --accept-data-loss \
  --url redis://localhost:6379

# Retry previously failed indexes
rvl migrate batch-resume \
  --state batch_state.yaml \
  --retry-failed \
  --accept-data-loss \
  --url redis://localhost:6379
```

`batch-resume` uses the `backup_dir` stored in `batch_state.yaml` unless you pass `--backup-dir` explicitly. If you pass a different directory for the same checkpoint, resume is rejected.

Note: If the batch plan involves quantization (e.g., float32 → float16), you must pass `--accept-data-loss` to `batch-resume`, just as with `batch-apply`. Omit `--accept-data-loss` if the batch plan does not involve quantization.

Checking Batch Status

```
rvl migrate batch-status --state batch_state.yaml
```

Example output:

```
Batch Migration Status
=====
Batch ID: batch_20260320_100000
Started: 2026-03-20T10:00:00Z
Updated: 2026-03-20T10:25:00Z

Completed: 2
- products_idx: success (10:02:30)
- users_idx: failed - Redis connection timeout (10:05:45)

In Progress: inventory_idx
Remaining: 1 (analytics_idx)
```

Batch Report

After completion, a `batch_report.yaml` is generated:

```
version: 1
batch_id: "batch_20260320_100000"
status: completed # or partial_failure, failed
summary:
  total_indexes: 3
  successful: 3
  failed: 0
  skipped: 0
  total_duration_seconds: 127.5
indexes:
- name: products_idx
  status: success
  report_path: ./reports/products_idx_report.yaml
- name: users_idx
  status: success
  report_path: ./reports/users_idx_report.yaml
- name: orders_idx
  status: success
  report_path: ./reports/orders_idx_report.yaml
completed_at: "2026-03-20T10:02:07Z"
```

Python API for Batch Migration

```

from redisvl.migration import BatchMigrationPlanner, BatchMigrationExecutor

# Create batch plan
planner = BatchMigrationPlanner()
batch_plan = planner.create_batch_plan(
    redis_url="redis://localhost:6379",
    pattern="*_idx",
    schema_patch_path="quantize_patch.yaml",
)

# Review applicability
for idx in batch_plan.indexes:
    if idx.applicable:
        print(f"Will migrate: {idx.name}")
    else:
        print(f"Skipping {idx.name}: {idx.skip_reason}")

# Execute batch
executor = BatchMigrationExecutor()
report = executor.apply(
    batch_plan,
    redis_url="redis://localhost:6379",
    state_path="batch_state.yaml",
    report_dir="./reports/",
    backup_dir="/tmp/migration_backups",
    progress_callback=lambda name, pos, total, status: print(f"[{pos}/{total}] {name}:
    ↳ {status}"),
)

print(f"Batch status: {report.status}")
print(f"Successful: {report.summary.successful}/{report.summary.total_indexes}")

```

Batch Migration Tips

1. **Test on a single index first:** Run a single-index migration to verify the patch works before applying to a batch.
2. **Use `continue_on_error` for large batches:** This ensures one failure doesn't block all remaining indexes.
3. **Schedule during low-traffic periods:** Each index has downtime during migration.
4. **Review skipped indexes:** The `skip_reason` often indicates schema differences that need attention.
5. **Keep state files:** The `batch_state.yaml` is essential for resume. Don't delete it until the batch completes successfully.

Performance Tuning

Batch Size

The `--batch-size` flag controls how many keys are read/written per Redis pipeline round-trip. The default of 500 is a good balance. Larger batches (1000+) reduce round-trips but increase per-batch memory and latency.

Backup Disk Space

For quantization migrations, original vectors are saved to `--backup-dir` before mutation. Approximate size: $\text{num_docs} \times \text{dims} \times \text{bytes_per_element}$.

Docs	Dims	Source dtype	Backup size
100K	768	float32	~292 MB
1M	768	float32	~2.9 GB
1M	1536	float32	~5.7 GB

HNSW vs FLAT Index Capacity

Note

When migrating from **HNSW** to **FLAT**, the target index may report a *higher* document count than the source. This is not a bug; it reflects a fundamental difference in how the two algorithms store vectors.

HNSW maintains a navigable small-world graph with per-node neighbor lists. This graph overhead limits how many vectors can fit in available memory. FLAT stores vectors as a simple array with no graph overhead.

If the source HNSW index was operating near its memory capacity, some documents may have been registered in Redis Search's document table but not fully indexed into the HNSW graph. After migration to FLAT, those same documents become fully searchable because FLAT requires less memory per vector.

The migration validator compares the total key count (`num_docs + hash_indexing_failures`) between source and target, so this scenario is handled correctly in the general case.

Learn more

- [Index Migrations](#): How migrations work and which changes are supported

Migrate an Index: Vector Quantization, Resume, Backup, and Wizard

Warning

The index migrator is an **experimental** feature. APIs, CLI commands, and on-disk formats (plans, backups) may change in future releases. Review migration plans carefully before applying them to production indexes.

This guide walks through a **vector quantization** migration (`float32` -> `float16`) end to end using the programmatic API. You will learn how to:

- Build a schema patch that describes the change
- Generate and review a migration **plan** (read-only)
- **Apply** the migration with a mandatory on-disk backup
- Find **where the backup lives** and inspect its progress header
- Understand **crash-safe resume** and safely re-run a migration
- **Reload original vectors from the backup** (rollback)
- Build and apply the same migration through the **wizard**

For conceptual background see [Index Migrations](#) and the [Migrate an Index how-to](#).

Prerequisites: a running Redis 8.0+ (or Redis Stack) at `redis://localhost:6379` and `redisvl` installed.

```
import os
import glob
import numpy as np
import yaml

from redisvl.index import SearchIndex
from redisvl.redis.utils import array_to_buffer
from redisvl.query import VectorQuery

REDIS_URL = "redis://localhost:6379"
INDEX_NAME = "products"
DIMS = 8
N_DOCS = 600

np.random.seed(42)

def delete_matching(client, pattern, batch_size=500):
    deleted = 0
    batch = []
    for key in client.scan_iter(match=pattern, count=batch_size):
        batch.append(key)
        if len(batch) >= batch_size:
            deleted += client.delete(*batch)
            batch = []
    if batch:
        deleted += client.delete(*batch)
    return deleted
```

1. Create a source index with float32 vectors

We start with a small Hash index whose embedding field stores full precision float32 vectors, then load some random data.

```
schema = {
    "index": {
        "name": INDEX_NAME,
        "prefix": "product",
```

(continues on next page)

(continued from previous page)

```

        "storage_type": "hash",
    },
    "fields": [
        {"name": "name", "type": "text"},
        {"name": "category", "type": "tag"},
        {
            "name": "embedding",
            "type": "vector",
            "attrs": {
                "algorithm": "flat",
                "dims": DIMS,
                "distance_metric": "cosine",
                "datatype": "float32",
            },
        },
    ],
],
}

index = SearchIndex.from_dict(schema, redis_url=REDIS_URL)
if index.exists():
    index.delete(drop=True)
stale_keys = delete_matching(index.client, "product:*")
if stale_keys:
    print(f"Removed {stale_keys} stale demo key(s)")
index.create()

vectors = np.random.rand(N_DOCS, DIMS).astype(np.float32)
data = [
    {
        "name": f"product {i}",
        "category": "electronics" if i % 2 == 0 else "books",
        "embedding": array_to_buffer(vectors[i], dtype="float32"),
    }
    for i in range(N_DOCS)
]
keys = index.load(data)
print(f"Loaded {len(keys)} documents into '{INDEX_NAME}'")

```

```
Loaded 600 documents into 'products'
```

```
!rvl index listall --url redis://localhost:6379
```

```
Indices:
1. products
```


2. Describe the change with a schema patch

A **schema patch** lists only what changes. Here we update the `embedding` field's datatype from `float32` to `float16` (a 2x memory reduction). We write it to a YAML file the planner can read.

```
patch = {
    "version": 1,
    "changes": {
        "update_fields": [
            {"name": "embedding", "attrs": {"datatype": "float16"}},
        ]
    },
}

with open("schema_patch.yaml", "w") as f:
    yaml.safe_dump(patch, f, sort_keys=False)

print(open("schema_patch.yaml").read())
```

```
version: 1
changes:
  update_fields:
    - name: embedding
      attrs:
        datatype: float16
```

3. Create a migration plan (read-only)

`create_plan` snapshots the live index, diffs it against the patch, and returns a `MigrationPlan`. **No data is modified.** Review the warnings and the classified changes before applying.

```
from redisvl.migration import MigrationPlanner, MigrationExecutor
from redisvl.migration.utils import write_yaml

planner = MigrationPlanner()
plan = planner.create_plan(
    index_name=INDEX_NAME,
    schema_patch_path="schema_patch.yaml",
    redis_url=REDIS_URL,
)

print("Index:      ", plan.source.index_name)
print("Mode:       ", plan.mode)
print("Requested:   ", plan.requested_changes)
print("Warnings:    ")
for w in plan.warnings:
    print("  -", w)

# Plans can be persisted to YAML and reloaded later (or via the CLI)
write_yaml(plan.model_dump(), "migration_plan.yaml")
print("\nSaved plan to migration_plan.yaml")
```

```

Index:      products
Mode:       drop_recreate
Requested:  {'version': 1, 'changes': {'add_fields': [], 'remove_fields': [], 'update_
↪fields': [{'name': 'embedding', 'attrs': {'datatype': 'float16'}, 'options': {}}],
↪'rename_fields': [], 'index': {}}}
Warnings:
  - Index downtime is required

Saved plan to migration_plan.yaml

```

4. Apply the migration with a mandatory backup

The executor requires `backup_dir` before applying any migration. For quantization, it writes original vectors to disk before mutating them. If you omit `backup_dir`, or pass a path that cannot be created or written, the migration fails before touching the index. The returned report records the resolved backup directory and any backup file prefixes used.

We also pass a `progress_callback` to watch each phase.

Note

This drops and recreates the index definition. Documents are preserved; only the index structure and vector encoding change. Pause writes during the migration window.

```

BACKUP_DIR = "./migration_backups"

# The migration does not start without a usable backup directory.
executor = MigrationExecutor()
try:
    executor.apply(plan, redis_url=REDIS_URL, backup_dir=None)
except ValueError as exc:
    print("Missing backup_dir is rejected before migration:", exc)

BAD_BACKUP_DIR = "./not_a_backup_dir"
if os.path.isdir(BAD_BACKUP_DIR):
    os.rmdir(BAD_BACKUP_DIR)
with open(BAD_BACKUP_DIR, "w") as f:
    f.write("this file intentionally blocks directory creation")
try:
    executor.apply(plan, redis_url=REDIS_URL, backup_dir=BAD_BACKUP_DIR)
except ValueError as exc:
    print("Invalid backup_dir is rejected before migration:", exc)
finally:
    if os.path.exists(BAD_BACKUP_DIR):
        os.remove(BAD_BACKUP_DIR)

def on_progress(step, detail=None):
    print(f"[{step}] {detail or ''}")

```

(continues on next page)

(continued from previous page)

```

report = executor.apply(
    plan,
    redis_url=REDIS_URL,
    backup_dir=BACKUP_DIR,
    batch_size=100,
    num_workers=1,
    progress_callback=on_progress,
)

print("\nResult:          ", report.result)
print("Total duration:    ", report.timings.total_migration_duration_seconds, "s")
print("Quantize duration:", report.timings.quantize_duration_seconds, "s")
print("Schema match:      ", report.validation.schema_match)
print("Doc count match:   ", report.validation.doc_count_match)
print("Backup dir:        ", report.backup.backup_dir)
print("Backup prefixes:   ", report.backup.backup_paths)

BACKUP_PREFIX = report.backup.backup_paths[0]

```

```

Missing backup_dir is rejected before migration: A backup directory is required to apply
→ migrations. Provide --backup-dir or backup_dir=...; migrations are not started without
→ a backup directory.
Invalid backup_dir is rejected before migration: Could not create or access backup
→ directory './not_a_backup_dir': [Errno 17] File exists: 'not_a_backup_dir'. A writable
→ backup directory is required to safely migrate.
[enumerate] Enumerating indexed documents...
[enumerate] found 600 documents (0.003s)
[dump] Backing up original vectors...
[dump] 100/600 docs
[dump] 200/600 docs
[dump] 300/600 docs
[dump] 400/600 docs
[dump] 500/600 docs
[dump] 600/600 docs
[dump] done (0.009s)
[drop] Dropping index definition...
[drop] done (0.001s)
[quantize] Re-encoding vectors from backup...
[quantize] 100/600 docs
[quantize] 200/600 docs
[quantize] 300/600 docs
[quantize] 400/600 docs
[quantize] 500/600 docs
[quantize] 600/600 docs
[quantize] done (600 docs in 0.009s)
[create] Creating index with new schema...

```

```

[create] done (0.004s)
[index] Waiting for re-indexing...
[index] 22/115 docs (19%)

```

```
[index] 600/600 docs (100%)
[index] done (0.508s)
[validate] Validating migration...
[validate] done (0.01s)

Result:          succeeded
Total duration:  0.554 s
Quantize duration: 0.009 s
Schema match:    True
Doc count match: True
Backup dir:       /Users/nitin.kanukolanu/workspace/redis-vl-python/docs/user_guide/
↳migration_backups
Backup prefixes:  ['/Users/nitin.kanukolanu/workspace/redis-vl-python/docs/user_guide/
↳migration_backups/migration_backup_products_0a3e27b8']
```

5. Where is the backup, and what's in it?

Backups are written under `report.backup.backup_dir`. For a single-worker Hash quantization migration there are two files per index:

- `migration_backup_<index>_<hash>.header` – JSON: phase + progress counters
- `migration_backup_<index>_<hash>.data` – binary: original vectors, batched

The `<hash>` suffix is a short digest of the index name, which avoids collisions. `report.backup.backup_paths` stores the path prefix without `.header` or `.data`. Multi-worker migrations record one prefix per worker.

Backups are **retained after success** so you can audit or roll back; delete them manually when no longer needed.

```
for path in sorted(glob.glob(os.path.join(report.backup.backup_dir, '*'))):
    size = os.path.getsize(path)
    print(f"{path} ({size:,} bytes)")
```

```
/Users/nitin.kanukolanu/workspace/redis-vl-python/docs/user_guide/migration_backups/
↳migration_backup_products_0a3e27b8.data (47,730 bytes)
/Users/nitin.kanukolanu/workspace/redis-vl-python/docs/user_guide/migration_backups/
↳migration_backup_products_0a3e27b8.header (209 bytes)
```

```
from redisvl.migration.backup import VectorBackup

# load() takes the path prefix, without .header or .data.
backup = VectorBackup.load(BACKUP_PREFIX)
h = backup.header
print("backup prefix:           ", BACKUP_PREFIX)
print("index_name:               ", h.index_name)
print("phase:                     ", h.phase)
print("batch_size:                 ", h.batch_size)
print("dump_completed_batches:    ", h.dump_completed_batches)
print("quantize_completed_batches:", h.quantize_completed_batches)
```

```
backup prefix:           /Users/nitin.kanukolanu/workspace/redis-vl-python/docs/user_
↳guide/migration_backups/migration_backup_products_0a3e27b8
```

(continues on next page)

(continued from previous page)

```

index_name:          products
phase:              completed
batch_size:         100
dump_completed_batches: 6
quantize_completed_batches: 6

```

6. Crash-safe resume and checkpointing

If a migration is interrupted by a crash, network drop, or Ctrl+C, just re-run the same command with the same `backup_dir`. The executor loads the backup header, validates that it belongs to the planned source index, and continues from the next unfinished batch.

The header is the checkpoint for single-index migrations:

- `phase` shows where the previous run stopped: `dump`, `ready`, `active`, or `completed`
- `dump_completed_batches` counts original-vector batches safely written to `.data`
- `quantize_completed_batches` counts batches already re-encoded and written back to Redis

```

# Re-running the same CLI command resumes automatically:
rvl migrate apply --plan migration_plan.yaml \
  --backup-dir ./migration_backups --url redis://localhost:6379

```

Batch migrations use the same per-index backup headers, plus a batch state YAML file that records the current index, completed indexes, failed indexes, and the `backup_dir`. Resume rejects a different backup directory so the checkpoint and backup files stay together.

When `phase` is `completed`, re-running is safe if the live index already matches the target schema: the executor detects the finished backup, skips completed work, and leaves the already-created index in place. If you have rolled back and the live index is back on the source schema, the old completed backup is stale for a new migration run; the executor discards that checkpoint and writes a fresh backup.

```

skip = backup.header.quantize_completed_batches
print(f"Checkpoint says {skip} batch(es) were already quantized.")
print(f"Current phase: {backup.header.phase}")

print("\nRe-running apply with the same backup_dir to exercise resume detection...")
resume_report = executor.apply(
    plan,
    redis_url=REDIS_URL,
    backup_dir=BACKUP_DIR,
    batch_size=100,
    num_workers=1,
    progress_callback=on_progress,
)
print("Resume result:      ", resume_report.result)
print("Resume backup dir: ", resume_report.backup.backup_dir)
print("Resume prefixes:   ", resume_report.backup.backup_paths)

```

```

Checkpoint says 6 batch(es) were already quantized.
Current phase: completed

```

(continues on next page)

(continued from previous page)

```

Re-running apply with the same backup_dir to exercise resume detection...
[enumerate] skipped (resume from backup)
[drop] skipped (already dropped)
[quantize] skipped (already completed)
[create] Creating index with new schema...
[create] done (0.004s)
[index] Waiting for re-indexing...
[index] 600/600 docs (100%)
[index] done (0.001s)
[validate] Validating migration...
[validate] done (0.008s)
Resume result:      succeeded
Resume backup dir:  /Users/nitin.kanukolanu/workspace/redis-vl-python/docs/user_guide/
↳migration_backups
Resume prefixes:    ['/Users/nitin.kanukolanu/workspace/redis-vl-python/docs/user_guide/
↳migration_backups/migration_backup_products_0a3e27b8']

```

7. Verify the quantized index

The documents were preserved and the embedding field is now float16. We reconnect to the live index and run a vector query (encoding the query vector to match the new datatype).

```

restored = SearchIndex.from_existing(INDEX_NAME, redis_url=REDIS_URL)
emb = next(f for f in restored.schema.to_dict()['fields'] if f['name'] == 'embedding')
print("embedding datatype now:", emb['attrs']['datatype'])

q = VectorQuery(
    vector=vectors[0].tolist(),
    vector_field_name="embedding",
    return_fields=["name", "category"],
    dtype="float16",
    num_results=3,
)
for r in restored.query(q):
    print(r["name"], "| category:", r["category"], "| dist:", r["vector_distance"])

```

```

embedding datatype now: float16
product 0 | category: electronics | dist: 0
product 223 | category: books | dist: 0.0458984375
product 23 | category: books | dist: 0.04736328125

```

8. Recover original vectors from the backup (rollback)

Because the backup holds the original float32 bytes, you can recover the pre-migration vector data. The CLI provides a one-liner:

```
rvl migrate rollback --backup-dir ./migration_backups \
  --index products --url redis://localhost:6379
```

Below is the equivalent **Python API**: iterate the backup batches and write the original bytes back with HSET. Rollback restores **data only**; afterwards recreate the original index definition so the index encoding matches the restored vectors again.

```
client = restored.client

restored_count = 0
for batch_keys, originals in backup.iter_batches():
    pipe = client.pipeline(transaction=False)
    for key in batch_keys:
        if key in originals:
            for field_name, original_bytes in originals[key].items():
                pipe.hset(key, field_name, original_bytes)
                restored_count += 1
    pipe.execute()

print(f"Restored original bytes for {restored_count} vector field(s)")

# Recreate the ORIGINAL float32 index definition over the restored data
original_index = SearchIndex.from_dict(schema, redis_url=REDIS_URL)
original_index.create(overwrite=True, drop=False)

check = SearchIndex.from_existing(INDEX_NAME, redis_url=REDIS_URL)
emb = next(f for f in check.schema.to_dict()['fields'] if f['name'] == 'embedding')
print("embedding datatype after rollback:", emb['attrs']['datatype'])
```

```
Restored original bytes for 600 vector field(s)
embedding datatype after rollback: float32
```

9. Build and apply a migration with the wizard

For exploratory work, MigrationWizard can build the same schema patch and migration plan interactively. In a notebook, we script the answers so the cell can execute without blocking. The sequence below means: update a field, choose embedding, keep the current algorithm, change datatype to float16, keep the distance metric, then finish.

The wizard still only creates the patch and plan. Applying the plan remains a separate reviewed step, and backup_dir is still required.

```
import builtins
import copy
from contextlib import contextmanager

from redisvl.migration import MigrationWizard
from redisvl.migration.utils import wait_for_index_ready
```

(continues on next page)

(continued from previous page)

```

WIZARD_INDEX_NAME = "wizard_products"
WIZARD_PREFIX = "wizard_product"
WIZARD_PATCH_PATH = "wizard_schema_patch.yaml"
WIZARD_PLAN_PATH = "wizard_migration_plan.yaml"
WIZARD_TARGET_SCHEMA_PATH = "wizard_target_schema.yaml"
WIZARD_BACKUP_DIR = "./wizard_migration_backups"

wizard_schema = copy.deepcopy(schema)
wizard_schema["index"]["name"] = WIZARD_INDEX_NAME
wizard_schema["index"]["prefix"] = WIZARD_PREFIX

# Start from a clean wizard demo index and keyspace.
try:
    existing_wizard_index = SearchIndex.from_existing(
        WIZARD_INDEX_NAME, redis_url=REDIS_URL
    )
    existing_wizard_index.delete(drop=True)
except Exception:
    pass
delete_matching(client, f"{WIZARD_PREFIX}:*")

wizard_index = SearchIndex.from_dict(wizard_schema, redis_url=REDIS_URL)
wizard_index.create()
wizard_index.load(data, id_field=None)
wait_for_index_ready(wizard_index)
print(f"Loaded {N_DOCS} documents into '{WIZARD_INDEX_NAME}'")

@contextmanager
def scripted_inputs(answers):
    original_input = builtins.input
    iterator = iter(answers)

    def fake_input(prompt=""):
        answer = next(iterator)
        print(f"{prompt}{answer}")
        return answer

    builtins.input = fake_input
    try:
        yield
    finally:
        builtins.input = original_input

wizard_answers = [
    "2",          # Update field
    "embedding",  # Select the vector field
    "",          # Keep algorithm
    "float16",    # Quantize datatype
    "",          # Keep distance metric

```

(continues on next page)

(continued from previous page)

```

    "8",          # Finish
]

with scripted_inputs(wizard_answers):
    wizard_plan = MigrationWizard().run(
        index_name=WIZARD_INDEX_NAME,
        redis_url=REDIS_URL,
        plan_out=WIZARD_PLAN_PATH,
        patch_out=WIZARD_PATCH_PATH,
        target_schema_out=WIZARD_TARGET_SCHEMA_PATH,
    )

print("\nWizard patch:")
print(open(WIZARD_PATCH_PATH).read())
print("Wizard plan mode:", wizard_plan.mode)
print("Wizard warnings:", wizard_plan.warnings)

```

```

Loaded 600 documents into 'wizard_products'
Building a migration plan for index 'wizard_products'
Current schema:
- Index name: wizard_products
- Storage type: hash
  - name (text)
  - category (tag)
  - embedding (vector)

Choose an action:
1. Add field      (text, tag, numeric, geo)
2. Update field   (sortable, weight, separator, vector config)
3. Remove field
4. Rename field   (rename field in all documents)
5. Rename index   (change index name)
6. Change prefix  (rename all keys)
7. Preview patch  (show pending changes as YAML)
8. Finish
Enter a number: 2
Updatable fields:
1. name (text)
2. category (tag)
3. embedding (vector)
Select a field to update by number or name: embedding
Current vector config for 'embedding':
  algorithm: FLAT
  datatype: float32
  distance_metric: cosine
  dims: 8 (cannot be changed)

Leave blank to keep current value.
  Algorithm: vector search method (FLAT=brute force, HNSW=graph, SVS-VAMANA=compressed_
↳ graph)
Algorithm [current: FLAT]:
  Datatype: float16, float32, bfloat16, float64, int8, uint8

```

(continues on next page)

(continued from previous page)

`(float16 reduces memory ~50%, int8/uint8 reduce ~75%)``Datatype [current: float32]: float16``Distance metric: how similarity is measured (cosine, l2, ip)``Distance metric [current: cosine]:``Choose an action:`

- `1. Add field (text, tag, numeric, geo)`
- `2. Update field (sortable, weight, separator, vector config)`
- `3. Remove field`
- `4. Rename field (rename field in all documents)`
- `5. Rename index (change index name)`
- `6. Change prefix (rename all keys)`
- `7. Preview patch (show pending changes as YAML)`
- `8. Finish`

`Enter a number: 8``Wizard patch:``version: 1``changes:`

```

  add_fields: []
  remove_fields: []
  update_fields:
  - name: embedding
    attrs:
      datatype: float16
      options: {}
  rename_fields: []
  index: {}

```

`Wizard plan mode: drop_recreate``Wizard warnings: ['Index downtime is required']`

```

wizard_report = MigrationExecutor().apply(
    wizard_plan,
    redis_url=REDIS_URL,
    backup_dir=WIZARD_BACKUP_DIR,
    batch_size=100,
    num_workers=1,
)

wizard_live = SearchIndex.from_existing(WIZARD_INDEX_NAME, redis_url=REDIS_URL)
wizard_embedding = next(
    f for f in wizard_live.schema.to_dict()["fields"] if f["name"] == "embedding"
)

print("Wizard migration result:", wizard_report.result)
print("Wizard schema match: ", wizard_report.validation.schema_match)
print("Wizard doc count match: ", wizard_report.validation.doc_count_match)
print("Wizard backup dir: ", wizard_report.backup.backup_dir)
print("Wizard backup prefixes: ", wizard_report.backup.backup_paths)
print("Wizard embedding dtype: ", wizard_embedding["attrs"]["datatype"])

```

```

Wizard migration result: succeeded
Wizard schema match:      True
Wizard doc count match:   True
Wizard backup dir:        /Users/nitin.kanukolanu/workspace/redis-vl-python/docs/user_
↳guide/wizard_migration_backups
Wizard backup prefixes:    ['/Users/nitin.kanukolanu/workspace/redis-vl-python/docs/user_
↳guide/wizard_migration_backups/migration_backup_wizard_products_def8cdf8']
Wizard embedding dtype:   float16

```

10. Cleanup (optional)

Remove the demo indexes and the artifacts this notebook created. In production, delete backups only once you are certain rollback is no longer needed.

```

delete_matching(client, "product:*")
if check.exists():
    check.delete(drop=False)

delete_matching(client, f"{WIZARD_PREFIX}:*")
if wizard_live.exists():
    wizard_live.delete(drop=False)

for backup_dir in (report.backup.backup_dir, wizard_report.backup.backup_dir):
    for f in glob.glob(os.path.join(backup_dir, '*')):
        os.remove(f)
    if os.path.isdir(backup_dir):
        os.rmdir(backup_dir)

for f in (
    "schema_patch.yaml",
    "migration_plan.yaml",
    "not_a_backup_dir",
    WIZARD_PATCH_PATH,
    WIZARD_PLAN_PATH,
    WIZARD_TARGET_SCHEMA_PATH,
):
    if os.path.exists(f):
        os.remove(f)
print("Cleaned up demo indexes, demo keys, backups, and YAML files")

```

```
Cleaned up demo indexes, demo keys, backups, and YAML files
```

Learn more

- *Migrate an Index (how-to)* – full CLI workflow, batch migration, performance tuning, and troubleshooting
- *Index Migrations (concepts)* – modes, supported vs blocked changes, backup internals, sync vs async
- For very large datasets, use `num_workers > 1` and the async executor (`AsyncMigrationExecutor`) to parallelize re-encoding.

1.2.4 The RedisVL CLI

RedisVL is a Python library with a dedicated CLI to create, inspect, list, migrate, and delete Redis search indexes, inspect index statistics, and run the RedisVL MCP server.

This notebook will walk through how to use the Redis Vector Library CLI (`rvl`).

Before running this notebook, be sure to

1. Have installed `redisvl` and have that environment active for this notebook.
2. Have a running Redis instance with Redis Search enabled

For complete command syntax and options, see the *CLI Reference*.

```
# First, see if the rvl tool is installed
!rvl version
```

Commands

The table below documents the current CLI tree. Use `rvl index --help` and `rvl stats --help` for detailed flag help and examples.

Command	Purpose
<code>rvl version</code>	display the installed RedisVL version
<code>rvl index create</code>	create a new Redis search index from a schema YAML file
<code>rvl index info</code>	display schema and storage details for an index
<code>rvl index listall</code>	list Redis search indexes available on the target Redis deployment
<code>rvl index delete</code>	delete an index while leaving indexed data in Redis
<code>rvl index destroy</code>	delete an index and drop its indexed data
<code>rvl stats</code>	display statistics for an existing Redis search index
<code>rvl mcp</code>	run the RedisVL MCP server
<code>rvl migrate wizard</code>	interactively build a migration plan and schema patch (experimental)
<code>rvl migrate plan</code>	generate <code>migration_plan.yaml</code> from a patch or target schema (experimental)
<code>rvl migrate apply</code>	execute a reviewed <code>drop_recreate</code> migration (experimental)
<code>rvl migrate validate</code>	validate a completed migration and emit report artifacts (experimental)
<code>rvl migrate rollback</code>	restore original vector bytes from a migration backup (experimental)
<code>rvl migrate batch-plan</code>	generate a batch plan for multiple indexes (experimental)
<code>rvl migrate batch-apply</code>	execute a batch migration with checkpoint state (experimental)
<code>rvl migrate batch-resume</code>	resume an interrupted batch migration (experimental)
<code>rvl migrate batch-status</code>	inspect batch migration checkpoint state (experimental)

Within data-plane commands, `-i` or `--index` targets an existing Redis index name and `-s` or `--schema` points to a schema YAML file. Shared Redis connection options such as `--url`, `--host`, and `--port` apply to `rvl index` and `rvl stats`.

Index

The `rvl index` command groups the index management workflows. Use `rvl index --help` to see the documented subcommands: `create`, `info`, `listall`, `delete`, and `destroy`. Whether you are working in Python or another language, this CLI can still be useful for managing and inspecting your indexes.

First, we will create an index from a yaml schema that looks like the following:

```
%%writefile schema.yaml

version: '0.1.0'

index:
  name: vectorizers
  prefix: doc
  storage_type: hash

fields:
  - name: sentence
    type: text
  - name: embedding
    type: vector
    attrs:
      dims: 768
      algorithm: flat
      distance_metric: cosine
```

Overwriting schema.yaml

```
# Create an index from a yaml schema
!rvl index create -s schema.yaml
```

Index created successfully

```
# list the indices that are available
!rvl index listall
```

Indices:
1. vectorizers

```
# inspect the index fields
!rvl index info -i vectorizers
```

Index Information:

Index Name	Storage Type	Prefixes	Index Options	Indexing
vectorizers	HASH	['doc']	[]	0

Index Fields:

Name	Attribute	Type	Field Option	Option Value
------	-----------	------	--------------	--------------

(continues on next page)

(continued from previous page)

↪	Field	Option	Option Value	Field	Option	Option Value	Field
↪	Option		Option Value				
↪	sentence		sentence	TEXT	WEIGHT	1	↪
↪							↪
↪	embedding		embedding	VECTOR	algorithm	FLAT	↪
↪	data_type		FLOAT32	dim	768	distance_	
↪	metric	COSINE					

```
# delete an index without deleting the data within it
!rvl index delete -i vectorizers
```

Index deleted successfully

```
# see the indices that still exist
!rvl index listall
```

Indices:

Stats

The `rvl stats` command returns basic information about an index. Use `-i` or `--index` to target an existing Redis index name, or `-s` or `--schema` to target a schema-defined index. Shared Redis connection options such as `--url`, `--host`, and `--port` also apply here.

```
# create a new index with the same schema
# recreating the index will reindex the documents
!rvl index create -s schema.yaml
```

Index created successfully

```
# list the indices that are available
!rvl index listall
```

Indices:
1. vectorizers

```
# see all the stats for the index
!rvl stats -i vectorizers
```

Statistics:

Stat Key	Value
num_docs	0
num_terms	0

(continues on next page)

(continued from previous page)

max_doc_id	0
num_records	0
percent_indexed	1
hash_indexing_failures	0
number_of_uses	1
bytes_per_record_avg	nan
doc_table_size_mb	0.00769805
inverted_sz_mb	0
key_table_size_mb	2.28881835
offset_bits_per_record_avg	nan
offset_vectors_sz_mb	0
offsets_per_term_avg	nan
records_per_doc_avg	nan
sortable_values_size_mb	0
total_indexing_time	0
total_inverted_index_blocks	0
vector_index_sz_mb	0

Migrate

The `rvl migrate` command provides a full workflow for changing index schemas without losing data. Common use cases include vector quantization (float32 → float16), algorithm changes (HNSW → FLAT), and adding/removing fields.

```
# List available indexes
rvl index listall --url redis://localhost:6379

# Build a migration plan interactively
rvl migrate wizard --index myindex --url redis://localhost:6379

# Or generate from a schema patch file
rvl migrate plan --index myindex --schema-patch patch.yaml --url redis://localhost:6379

# Apply with backup and multi-worker quantization
rvl migrate apply --plan migration_plan.yaml --url redis://localhost:6379 \
  --backup-dir /tmp/backups --workers 4 --batch-size 500

# Validate the result
rvl migrate validate --plan migration_plan.yaml --url redis://localhost:6379
```

See the *Migration Guide* for detailed usage, performance tuning, and examples.

Optional arguments

You can modify these commands with the below optional arguments

Argument	Description	Default
-u --url	The full Redis URL to connect to	redis://localhost:6379
--host	Redis host to connect to	localhost
-p --port	Redis port to connect to. Must be an integer	6379
--user	Redis username, if one is required	default
--ssl	Boolean flag indicating if ssl is required. If set the Redis base url changes to rediss://	None
-a --password	Redis password, if one is required	""

Choosing your Redis instance

By default rvl first checks if you have REDIS_URL environment variable defined and tries to connect to that. If not, it then falls back to localhost:6379, unless you pass the --host or --port arguments

```
# specify your Redis instance to connect to
!rvl index listall --host localhost --port 6379
```

```
# NBVAL_SKIP
# Not run in CI: there is no TLS-enabled Redis to connect to, so the client
# blocks on the handshake until the nbval cell timeout expires.
# connect to rediss://jane_doe:password123@localhost:6379
!rvl index listall --user jane_doe -a password123 --ssl
```

```
!rvl index destroy -i vectorizers
```

1.2.5 Use Cases

RedisVL powers a wide range of AI applications. Here's how to apply its features to common use cases.

Agent Context Provide agents with the right information at the right time.

- **RAG** – Retrieval-Augmented Generation with *vector search* and *hybrid queries*
- **Memory** – Persistent *message history* across sessions
- **Context Engineering** – Combine *filtering*, *reranking*, and *embeddings* to curate the optimal context window

Agent Optimization Reduce latency and cost for AI workloads.

- **Semantic Caching** – Cache LLM responses by meaning with *SemanticCache*
- **Embeddings Caching** – Avoid redundant embedding calls with *EmbeddingsCache*
- **Semantic Routing** – Route queries to the right handler with *SemanticRouter*

General Search Build search experiences that understand meaning, not just keywords.

- **Semantic Search** – *Vector queries* with *complex filtering*

- **Hybrid Search** – Combine keyword and vector search with *advanced query types*
- **SQL Translation** – Use familiar SQL syntax with *SQLQuery*

Personalization & RecSys Drive engagement with personalized recommendations.

- **User Similarity** – Find similar users or items using *vector search*
- **Real-Time Ranking** – Combine vector similarity with *metadata filtering* and *reranking*
- **Multi-Signal Matching** – Search across multiple embedding fields with *MultiVectorQuery*

1.3 Example Gallery

Explore community examples of RedisVL in the wild.

 Tip

For a comprehensive collection of Redis AI examples, tutorials, and recipes, visit the [Redis AI Resources](#) repository. It includes notebooks for RAG, agents, semantic caching, recommendation systems, and more.

1.3.1 Demo Applications

Full-stack applications showcasing RedisVL and Redis vector search capabilities.

Redis RAG Workbench	Arxiv Paper Search	ArxivChatGuru	eCommerce Search
Redis Movies Searcher (Java)	Agentic RAG	Shopping AI Agent	Restaurant Discovery
AI Agent	Podcast Chatbot with Agent Memory	Car Dealership Assistant	Multi-Agent
Text Adventure Game	Memory-Aware Alexa Assistant	Virtual Banking Assistant	LLM
Recommender for Hotels	Real-Time Embeddings with Bytewax	Redis VSS Streamlit Demo	

 Note

If you are using RedisVL, please consider adding your example to this page by opening a Pull Request on [GitHub](#)

1.3.2 Code Recipes

Runnable Jupyter notebooks from [Redis AI Resources](#) covering real-world use cases. Each recipe includes a Google Colab link for easy execution.

RAG with Frameworks

Build retrieval-augmented generation pipelines with popular frameworks.

Recipe	Description	Links
RAG with LangChain	RAG using Redis and LangChain	GitHub Colab
RAG with LlamaIndex	RAG using Redis and LlamaIndex	GitHub Colab
Advanced RAG	Advanced RAG techniques with RedisVL	GitHub Colab
NVIDIA RAG	RAG using Redis and NVIDIA NIMs	GitHub Colab
RAGAS Evaluation	Evaluate RAG performance with RAGAS	GitHub Colab
Role-Based RAG	Implement RBAC policies with vector search	GitHub Colab

Agents

Build AI agents with memory, tools, and multi-agent workflows.

Recipe	Description	Links
LangGraph Agents	Get started with LangGraph and agentic RAG	GitHub Colab
CrewAI Agents	Multi-agent systems with CrewAI and LangGraph	GitHub Colab
Full-Featured Agent	Tool-calling agent with semantic cache and router	GitHub Colab
Memory Agent	Agent with short-term and long-term memory	GitHub Colab
Autogen Agent	Blog writing agent with Autogen and Redis	GitHub Colab

Recommendation Systems

Build personalized recommendation engines with Redis.

Recipe	Description	Links
Content Filtering	Content-based filtering with RedisVL	GitHub Colab
Collaborative Filtering	Collaborative filtering with RedisVL	GitHub Colab
Two Towers	Deep learning two-tower model with RedisVL	GitHub Colab

Specialized Applications

Explore Redis for computer vision, feature stores, and AI gateways.

Recipe	Description	Links
Facial Recognition	Build a facial recognition system with Facenet and RedisVL	GitHub Colab
Credit Scoring	Credit scoring with Feast and Redis as online store	GitHub Colab
Transaction Search	Real-time transaction feature search	GitHub Colab
LiteLLM Gateway	Getting started with LiteLLM proxy and Redis	GitHub Colab

Vector Search Deep Dives

Advanced vector search techniques and optimizations.

Recipe	Description	Links
Vector Search with redis-py	Low-level vector search with Redis Python client	GitHub Colab
Hybrid Search	Combine BM25 and vector search	GitHub Colab
Data Type Support	Convert float32 index to float16 or integer	GitHub Colab
Benchmarking Basics	Search benchmarking with RedisVL	GitHub Colab
Multi-Vector Search	Multi-vector queries with RedisVL	GitHub Colab
HNSW to SVS-VAMANA Migration	Migrate HNSW indices to SVS-VAMANA	GitHub Colab

LLM Optimization

Reduce costs and latency with caching and routing.

Recipe	Description	Links
Gemini Semantic Cache	Semantic caching with Redis and Google Gemini	GitHub Colab
Doc2Cache with Llama3.1	Semantic caching with Doc2Cache framework	GitHub Colab
Cache Optimization	Optimize cache thresholds with redis-retrieval-optimizer	GitHub Colab
Context-Enabled Caching	Context-aware semantic caching	GitHub Colab
Router Optimization	Optimize router thresholds	GitHub Colab

1.3.3 More Resources

Looking for more examples and tutorials?

- **Redis AI Resources** – Comprehensive collection of code recipes, demos, and tutorials
- **Java Recipes** – Spring AI, Redis OM Spring, and semantic routing examples
- **Redis Developer Hub** – Official Redis developer resources

1.4 RedisVL API

Reference documentation for the RedisVL API.

1.4.1 Schema

Schema in RedisVL provides a structured format to define index settings and field configurations using the following three components:

Component	Description
<i>version</i>	The version of the schema spec. Current supported version is <i>0.1.0</i> .
<i>index</i>	Index specific settings like name, key prefix, key separator, and storage type.
<i>fields</i>	Subset of fields within your data to include in the index and any custom settings.

IndexSchema

class IndexSchema(*, index, fields=<factory>, version='0.1.0')

A schema definition for a search index in Redis, used in RedisVL for configuring index settings and organizing vector and metadata fields.

The class offers methods to create an index schema from a YAML file or a Python dictionary, supporting flexible schema definitions and easy integration into various workflows.

An example *schema.yaml* file might look like this:

```
version: '0.1.0'

index:
  name: user-index
  prefix: user
  key_separator: ":"
  storage_type: json

fields:
  - name: user
    type: tag
  - name: credit_score
    type: tag
  - name: embedding
    type: vector
    attrs:
      algorithm: flat
      dims: 3
      distance_metric: cosine
      datatype: float32
```

Loading the schema for RedisVL from yaml is as simple as:

```
from redisvl.schema import IndexSchema

schema = IndexSchema.from_yaml("schema.yaml")
```

Loading the schema for RedisVL from dict is as simple as:

```
from redisvl.schema import IndexSchema

schema = IndexSchema.from_dict({
    "index": {
        "name": "user-index",
        "prefix": "user",
        "key_separator": ":",
        "storage_type": "json",
    },
    "fields": [
        {"name": "user", "type": "tag"},
        {"name": "credit_score", "type": "tag"},
        {
            "name": "embedding",
            "type": "vector",
        }
    ]
})
```

(continues on next page)

(continued from previous page)

```

        "attrs": {
            "algorithm": "flat",
            "dims": 3,
            "distance_metric": "cosine",
            "datatype": "float32"
        }
    }
]
})

```

Note

The *fields* attribute in the schema must contain unique field names to ensure correct and unambiguous field references.

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **index** (*IndexInfo*)
- **fields** (*dict[str, BaseField]*)
- **version** (*Literal['0.1.0']*)

add_field(*field_inputs*)

Adds a single field to the index schema based on the specified field type and attributes.

This method allows for the addition of individual fields to the schema, providing flexibility in defining the structure of the index.

Parameters

field_inputs (*Dict[str, Any]*) – A field to add.

Raises

ValueError – If the field name or type are not provided or if the name already exists within the schema.

```

# Add a tag field
schema.add_field({"name": "user", "type": "tag"})

# Add a vector field
schema.add_field({
    "name": "user-embedding",
    "type": "vector",
    "attrs": {
        "dims": 1024,
        "algorithm": "flat",
        "datatype": "float32"
    }
})

```

add_fields(*fields*)

Extends the schema with additional fields.

This method allows dynamically adding new fields to the index schema. It processes a list of field definitions.

Parameters

fields (*List[Dict[str, Any]]*) – A list of fields to add.

Raises

ValueError – If a field with the same name already exists in the schema.

```
schema.add_fields([
    {"name": "user", "type": "tag"},
    {"name": "bio", "type": "text"},
    {
        "name": "user-embedding",
        "type": "vector",
        "attrs": {
            "dims": 1024,
            "algorithm": "flat",
            "datatype": "float32"
        }
    }
])
```

classmethod from_dict(*data*)

Create an IndexSchema from a dictionary.

Parameters

data (*Dict[str, Any]*) – The index schema data.

Returns

The index schema.

Return type

IndexSchema

```
from redisvl.schema import IndexSchema

schema = IndexSchema.from_dict({
    "index": {
        "name": "docs-index",
        "prefix": "docs",
        "storage_type": "hash",
    },
    "fields": [
        {
            "name": "doc-id",
            "type": "tag"
        },
        {
            "name": "doc-embedding",
            "type": "vector",
            "attrs": {
                "algorithm": "flat",
```

(continues on next page)

(continued from previous page)

```

        "dims": 1536
    }
}
]
})

```

classmethod `from_yaml(file_path)`

Create an IndexSchema from a YAML file.

Parameters

file_path (*str*) – The path to the YAML file.

Returns

The index schema.

Return type

IndexSchema

```

from redisvl.schema import IndexSchema
schema = IndexSchema.from_yaml("schema.yaml")

```

remove_field(field_name)

Removes a field from the schema based on the specified name.

This method is useful for dynamically altering the schema by removing existing fields.

Parameters

field_name (*str*) – The name of the field to be removed.

to_dict()

Serialize the index schema model to a dictionary, handling Enums and other special cases properly.

Returns

The index schema as a dictionary.

Return type

`Dict[str, Any]`

to_yaml(file_path, overwrite=True)

Write the index schema to a YAML file.

Parameters

- **file_path** (*str*) – The path to the YAML file.
- **overwrite** (*bool*) – Whether to overwrite the file if it already exists.

Raises

FileExistsError – If the file already exists and `overwrite` is `False`.

Return type

`None`

property field_names: list[str]

A list of field names associated with the index schema.

Returns

A list of field names from the schema.

Return type

List[str]

fields: Dict[str, BaseField]

Fields associated with the search index and their properties.

Note: When creating from dict/YAML, provide fields as a list of field definitions. The validator will convert them to a Dict[str, BaseField] internally.

index: IndexInfo

Details of the basic index configurations.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

version: Literal['0.1.0']

Version of the underlying index schema.

Index-Level Stopwords Configuration

The IndexInfo class supports index-level stopwords configuration through the `stopwords` field. This controls which words are filtered during indexing (server-side), as opposed to query-time filtering (client-side).

Configuration Options:

- None (default): Use Redis default stopwords (~300 common words)
- [] (empty list): Disable stopwords completely (STOPWORDS 0)
- Custom list: Specify your own stopwords (e.g., ["the", "a", "an"])

Example:

```
from redisvl.schema import IndexSchema

# Disable stopwords to search for phrases like "Bank of Glasberliner"
schema = IndexSchema.from_dict({
    "index": {
        "name": "company-idx",
        "prefix": "company",
        "stopwords": [] # STOPWORDS 0
    },
    "fields": [
        {"name": "name", "type": "text"}
    ]
})
```

Important Notes:

- Index-level stopwords affect what gets indexed (server-side)
- Query-time stopwords (in TextQuery and AggregateHybridQuery) affect what gets searched (client-side)
- Using query-time stopwords with index-level STOPWORDS 0 is counterproductive

For detailed information about stopwords configuration and best practices, see the Advanced Queries user guide (docs/user_guide/11_advanced_queries.ipynb).

Defining Fields

Fields in the schema can be defined in YAML format or as a Python dictionary, specifying a name, type, an optional path, and attributes for customization.

YAML Example:

```
- name: title
  type: text
  path: $.document.title
  attrs:
    weight: 1.0
    no_stem: false
    withsuffixtrie: true
```

Python Dictionary Example:

```
{
  "name": "location",
  "type": "geo",
  "attrs": {
    "sortable": true
  }
}
```

Basic Field Types

RedisVL supports several basic field types for indexing different kinds of data. Each field type has specific attributes that customize its indexing and search behavior.

Text Fields

Text fields support full-text search with stemming, phonetic matching, and other text analysis features.

class `TextField`(*, name, type=`FieldTypes.TEXT`, path=`None`, attrs=<factory>)

Bases: `BaseField`

Text field supporting a full text search index

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (*Literal[FieldTypes.TEXT]*)
- **path** (*str | None*)
- **attrs** (*TextFieldAttributes*)

as_redis_field()

Convert schema field to Redis Field object

Return type

Field

attrs: *TextFieldAttributes*

Specified field attributes

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

type: `Literal[FieldTypes.TEXT]`

Field type

```
class TextFieldAttributes(*, sortable=False, index_missing=False, no_index=False, weight=1,
                        no_stem=False, withsuffixtrie=False, phonetic_matcher=None,
                        index_empty=False, unf=False)
```

Full text field attributes

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **sortable** (*bool*)
- **index_missing** (*bool*)
- **no_index** (*bool*)
- **weight** (*float*)
- **no_stem** (*bool*)
- **withsuffixtrie** (*bool*)
- **phonetic_matcher** (*str | None*)
- **index_empty** (*bool*)
- **unf** (*bool*)

index_empty: `bool`

Allow indexing and searching for empty strings

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

no_stem: `bool`

Disable stemming on the text field during indexing

phonetic_matcher: `str | None`

Used to perform phonetic matching during search

unf: `bool`

Un-normalized form - disable normalization on sortable fields (only applies when `sortable=True`)

weight: float

Declares the importance of this field when calculating results

withsuffixtrie: bool

Keep a suffix trie with all terms which match the suffix to optimize certain queries

Tag Fields

Tag fields are optimized for exact-match filtering and faceted search on categorical data.

class TagField(**, name, type=FieldTypes.TAG, path=None, attrs=<factory>*)

Bases: BaseField

Tag field for simple boolean-style filtering

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (*Literal[FieldTypes.TAG]*)
- **path** (*str | None*)
- **attrs** (*TagFieldAttributes*)

as_redis_field()

Convert schema field to Redis Field object

Return type

Field

attrs: *TagFieldAttributes*

Specified field attributes

model_config: *ClassVar[ConfigDict] = {}*

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

type: *Literal[FieldTypes.TAG]*

Field type

class TagFieldAttributes(**, sortable=False, index_missing=False, no_index=False, separator=',', case_sensitive=False, withsuffixtrie=False, index_empty=False*)

Tag field attributes

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **sortable** (*bool*)

- **index_missing** (*bool*)
- **no_index** (*bool*)
- **separator** (*str*)
- **case_sensitive** (*bool*)
- **withsuffixtrie** (*bool*)
- **index_empty** (*bool*)

case_sensitive: **bool**

Treat text as case sensitive or not. By default, tag characters are converted to lowercase

index_empty: **bool**

Allow indexing and searching for empty strings

model_config: **ClassVar**[**ConfigDict**] = {}

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

separator: **str**

Indicates how the text in the original attribute is split into individual tags

withsuffixtrie: **bool**

Keep a suffix trie with all terms which match the suffix to optimize certain queries

Numeric Fields

Numeric fields support range queries and sorting on numeric data.

class **NumericField**(**, name, type=FieldTypes.NUMERIC, path=None, attrs=<factory>*)

Bases: **BaseField**

Numeric field for numeric range filtering

Create a new model by parsing and validating input data from keyword arguments.

Raises *[ValidationError][pydantic_core.ValidationError]* if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (*Literal[FieldTypes.NUMERIC]*)
- **path** (*str | None*)
- **attrs** (*NumericFieldAttributes*)

as_redis_field()

Convert schema field to Redis Field object

Return type

Field

attrs: *NumericFieldAttributes*

Specified field attributes

```
model_config: ClassVar[ConfigDict] = {}
```

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

```
type: Literal[FieldTypes.NUMERIC]
```

Field type

```
class NumericFieldAttributes(*, sortable=False, index_missing=False, no_index=False, unf=False)
```

Numeric field attributes

Create a new model by parsing and validating input data from keyword arguments.

Raises [ValidationError][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **sortable** (*bool*)
- **index_missing** (*bool*)
- **no_index** (*bool*)
- **unf** (*bool*)

```
model_config: ClassVar[ConfigDict] = {}
```

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

```
unf: bool
```

Un-normalized form - disable normalization on sortable fields (only applies when sortable=True)

Geo Fields

Geo fields enable location-based search with geographic coordinates.

```
class GeoField(*, name, type=FieldTypes.GEO, path=None, attrs=<factory>)
```

Bases: BaseField

Geo field with a geo-spatial index for location based search

Create a new model by parsing and validating input data from keyword arguments.

Raises [ValidationError][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (*Literal[FieldTypes.GEO]*)
- **path** (*str | None*)
- **attrs** (*GeoFieldAttributes*)

as_redis_field()

Convert schema field to Redis Field object

Return type

Field

attrs: *GeoFieldAttributes*

Specified field attributes

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

type: `Literal[FieldTypes.GEO]`

Field type

class GeoFieldAttributes(**, sortable=False, index_missing=False, no_index=False*)

Numeric field attributes

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **sortable** (*bool*)
- **index_missing** (*bool*)
- **no_index** (*bool*)

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

Vector Field Types

Vector fields enable semantic similarity search using various algorithms. All vector fields share common attributes but have algorithm-specific configurations.

Common Vector Attributes

All vector field types share these base attributes:

class BaseVectorFieldAttributes(**, dims, algorithm, datatype=VectorDataType.FLOAT32, distance_metric=VectorDistanceMetric.COSINE, initial_cap=None, index_missing=False*)

Base vector field attributes shared by FLAT, HNSW, and SVS-VAMANA fields

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **dims** (*int*)
- **algorithm** (*VectorIndexAlgorithm*)
- **datatype** (*VectorDataType*)
- **distance_metric** (*VectorDistanceMetric*)
- **initial_cap** (*int | None*)
- **index_missing** (*bool*)

classmethod uppercase_strings(*v*)

Validate that provided values are cast to uppercase

algorithm: **VectorIndexAlgorithm**

FLAT, HNSW, or SVS-VAMANA

Type

The indexing algorithm for the field

datatype: **VectorDataType**

The float datatype for the vector embeddings

dims: **int**

Dimensionality of the vector embeddings field

distance_metric: **VectorDistanceMetric**

The distance metric used to measure query relevance

property field_data: **dict[str, Any]**

Select attributes required by the Redis API

index_missing: **bool**

Allow indexing and searching for missing values (documents without the field)

initial_cap: **int | None**

Initial vector capacity in the index affecting memory allocation size of the index

model_config: **ClassVar[ConfigDict] = {}**

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

Key Attributes:

- *dims*: Dimensionality of the vector (e.g., 768, 1536).
- *algorithm*: Indexing algorithm for vector search:
 - *flat*: Brute-force exact search. 100% recall, slower for large datasets. Best for <10K vectors.
 - *hns**w*: Graph-based approximate search. Fast with high recall (95-99%). Best for general use.
 - *svs-vamana*: SVS-VAMANA (Scalable Vector Search with VAMANA graph algorithm) provides fast approximate nearest neighbor search with optional compression support. This algorithm is optimized for Intel hardware and offers reduced memory usage through vector compression.

Note

For detailed algorithm comparison and selection guidance, see *Vector Algorithm Comparison*.

- *datatype*: Float precision (*bfloat16*, *float16*, *float32*, *float64*). Note: SVS-VAMANA only supports *float16* and *float32*.
- *distance_metric*: Similarity metric (*COSINE*, *L2*, *IP*).
- *initial_cap*: Initial capacity hint for memory allocation (optional).
- *index_missing*: When True, allows searching for documents missing this field (optional).

HNSW Vector Fields

HNSW (Hierarchical Navigable Small World) - Graph-based approximate search with excellent recall. **Best for general-purpose vector search (10K-1M+ vectors).**

When to use HNSW & Performance Details

Use HNSW when:

- Medium to large datasets (100K-1M+ vectors) requiring high recall rates
- Search accuracy is more important than memory usage
- Need general-purpose vector search with balanced performance
- Cross-platform deployments where hardware-specific optimizations aren't available

Performance characteristics:

- **Search speed**: Very fast approximate search with tunable accuracy (via *ef_runtime* at query time)
- **Memory usage**: Higher than compressed SVS-VAMANA but reasonable for most applications
- **Recall quality**: Excellent recall rates (95-99%), tunable via *ef_runtime* parameter
- **Build time**: Moderate construction time, faster than SVS-VAMANA for smaller datasets

Runtime parameters (adjustable at query time without rebuilding index):

- *ef_runtime*: Controls search accuracy (higher = better recall, slower search). Default: 10
- *epsilon*: Range search approximation factor for *VectorRangeQuery*. Default: 0.01

class `HNSWVectorField`(*, *name*, *type*='vector', *path*=None, *attrs*)

Bases: `BaseField`

Vector field with HNSW (Hierarchical Navigable Small World) indexing for approximate nearest neighbor search.

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (*Literal*['vector'])
- **path** (*str* | None)
- **attrs** (`HNSWVectorFieldAttributes`)

as_redis_field()

Convert schema field to Redis Field object

Return type

Field

attrs: *HNSWVectorFieldAttributes*

Specified field attributes

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

type: `Literal['vector']`

Field type

```
class HNSWVectorFieldAttributes(* , dims, algorithm=VectorIndexAlgorithm.HNSW,
                                datatype=VectorDataType.FLOAT32,
                                distance_metric=VectorDistanceMetric.COSINE, initial_cap=None,
                                index_missing=False, m=16, ef_construction=200, ef_runtime=10,
                                epsilon=0.01)
```

HNSW vector field attributes for approximate nearest neighbor search.

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **dims** (*int*)
- **algorithm** (*Literal[VectorIndexAlgorithm.HNSW]*)
- **datatype** (*VectorDataType*)
- **distance_metric** (*VectorDistanceMetric*)
- **initial_cap** (*int | None*)
- **index_missing** (*bool*)
- **m** (*int*)
- **ef_construction** (*int*)
- **ef_runtime** (*int*)
- **epsilon** (*float*)

algorithm: `Literal[VectorIndexAlgorithm.HNSW]`

The indexing algorithm (fixed as 'hnsw')

ef_construction: `int`

100-800)

Type

Max edge candidates during build time (default

Type

200, range

ef_runtime: int

- 10) • primary tuning parameter

Type

Max top candidates during search (default

epsilon: float

0.01)

Type

Range search boundary factor (default

m: int

8-64)

Type

Max outgoing edges per node in each layer (default

Type

16, range

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

HNSW Examples:

Balanced configuration (recommended starting point):

```
- name: embedding
  type: vector
  attrs:
    algorithm: hnsw
    dims: 768
    distance_metric: cosine
    datatype: float32
    # Index-time parameters (set during index creation)
    m: 16 # Graph connectivity
    ef_construction: 200 # Build-time accuracy
    # Note: ef_runtime can be set at query time via VectorQuery
```

High-recall configuration:

```
- name: embedding
  type: vector
  attrs:
    algorithm: hnsw
    dims: 768
    distance_metric: cosine
    datatype: float32
    # Index-time parameters tuned for maximum accuracy
    m: 32
    ef_construction: 400
    # Note: ef_runtime=50 can be set at query time for higher recall
```

SVS-VAMANA Vector Fields

SVS-VAMANA (Scalable Vector Search with VAMANA graph algorithm) provides fast approximate nearest neighbor search with optional compression support. This algorithm is optimized for Intel hardware and offers reduced memory usage through vector compression. **Best for large datasets (>100K vectors) on Intel hardware with memory constraints.**

When to use SVS-VAMANA & Detailed Guide

Requirements:

- Redis $\geq$ 8.2.0 with Redis Search $\geq$ 2.8.10
- datatype must be 'float16' or 'float32' (float64/bfloat16 not supported)

Use SVS-VAMANA when:

- Large datasets where memory is expensive
- Cloud deployments with memory-based pricing
- When 90-95% recall is acceptable
- High-dimensional vectors (>1024 dims) with LeanVec compression

Performance vs other algorithms:

- **vs FLAT:** Much faster search, significantly lower memory usage with compression, but approximate results
- **vs HNSW:** Better memory efficiency with compression, similar or better recall, Intel-optimized

Runtime parameters (adjustable at query time without rebuilding index):

- **epsilon:** Range search approximation factor. Default: 0.01
- **search_window_size:** Size of search window for KNN searches. Higher = better recall, slower search
- **use_search_history:** Whether to use search buffer (OFF/ON/AUTO). Default: AUTO
- **search_buffer_capacity:** Tuning parameter for 2-level compression. Default: search_window_size

Compression selection guide:

- **No compression:** Best performance, standard memory usage
- **LVQ4/LVQ8:** Good balance of compression (2x-4x) and performance
- **LeanVec4x8/LeanVec8x8:** Maximum compression (up to 8x) with dimensionality reduction

Memory Savings Examples (1M vectors, 768 dims):

- No compression (float32): 3.1 GB
- LVQ4x4 compression: 1.6 GB (~48% savings)
- LeanVec4x8 + reduce to 384: 580 MB (~81% savings)

class SVSVectorField(*, name, type=*FieldTypes.VECTOR*, path=None, attrs)

Bases: BaseField

Vector field with SVS-VAMANA indexing and compression for memory-efficient approximate nearest neighbor search.

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (*Literal[FieldTypes.VECTOR]*)
- **path** (*str | None*)
- **attrs** (*SVSVectorFieldAttributes*)

as_redis_field()

Convert schema field to Redis Field object

Return type

Field

attrs: *SVSVectorFieldAttributes*

Specified field attributes

model_config: *ClassVar[ConfigDict] = {}*

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

type: *Literal[FieldTypes.VECTOR]*

Field type

```
class SVSVectorFieldAttributes(*, dims, algorithm=VectorIndexAlgorithm.SVS_VAMANA,
                             datatype=VectorDataType.FLOAT32,
                             distance_metric=VectorDistanceMetric.COSINE, initial_cap=None,
                             index_missing=False, graph_max_degree=40,
                             construction_window_size=250, search_window_size=20, epsilon=0.01,
                             compression=None, reduce=None, training_threshold=None)
```

SVS-VAMANA vector field attributes with compression support.

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **dims** (*int*)
- **algorithm** (*Literal[VectorIndexAlgorithm.SVS_VAMANA]*)
- **datatype** (*VectorDataType*)
- **distance_metric** (*VectorDistanceMetric*)
- **initial_cap** (*int | None*)
- **index_missing** (*bool*)
- **graph_max_degree** (*int*)
- **construction_window_size** (*int*)
- **search_window_size** (*int*)

- **epsilon** (*float*)
- **compression** (*CompressionType* | *None*)
- **reduce** (*int* | *None*)
- **training_threshold** (*int* | *None*)

validate_svs_params()

Validate SVS-VAMANA specific constraints

algorithm: `Literal[VectorIndexAlgorithm.SVS_VAMANA]`

The indexing algorithm for the vector field

compression: `CompressionType` | `None`

LVQ4, LVQ8, LeanVec4x8, LeanVec8x8

Type

Vector compression

construction_window_size: `int`

250) • affects quality vs build time

Type

Build-time candidates (default

epsilon: `float`

0.01)

Type

Range query boundary factor (default

graph_max_degree: `int`

40) • affects recall vs memory

Type

Max edges per node (default

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

reduce: `int` | `None`

Dimensionality reduction for LeanVec types (must be < dims)

search_window_size: `int`

20) • primary tuning parameter

Type

Search candidates (default

training_threshold: `int` | `None`

10,240)

Type

Min vectors before compression training (default

SVS-VAMANA Examples:**Basic configuration (no compression):**

```
- name: embedding
  type: vector
  attrs:
    algorithm: svsvamana
    dims: 768
    distance_metric: cosine
    datatype: float32
    # Index-time parameters (set during index creation)
    graph_max_degree: 40
    construction_window_size: 250
    # Note: search_window_size and other runtime params can be set at query time
```

High-performance configuration with compression:

```
- name: embedding
  type: vector
  attrs:
    algorithm: svsvamana
    dims: 768
    distance_metric: cosine
    datatype: float32
    # Index-time parameters tuned for better recall
    graph_max_degree: 64
    construction_window_size: 500
    # Maximum compression with dimensionality reduction
    compression: LeanVec4x8
    reduce: 384 # 50% dimensionality reduction
    training_threshold: 1000
    # Note: search_window_size=40 can be set at query time for higher recall
```

Important Notes:

- **Requirements:** SVS-VAMANA requires Redis ≥ 8.2 with Redis Search $\geq 2.8.10$.
- **Datatype limitations:** SVS-VAMANA only supports *float16* and *float32* datatypes (not *bfloat16* or *float64*).
- **Compression compatibility:** The *reduce* parameter is only valid with LeanVec compression types (*LeanVec4x8* or *LeanVec8x8*).
- **Platform considerations:** Intel's proprietary LVQ and LeanVec optimizations are not available in Redis Open Source. On non-Intel platforms and Redis Open Source, SVS-VAMANA with compression falls back to basic 8-bit scalar quantization.
- **Performance tip:** Runtime parameters like *search_window_size*, *epsilon*, and *use_search_history* can be adjusted at query time without rebuilding the index. Start with defaults and tune *search_window_size* first for your speed vs accuracy requirements.

FLAT Vector Fields

FLAT - Brute-force exact search. **Best for small datasets (<10K vectors) requiring 100% accuracy.**

When to use FLAT & Performance Details

Use FLAT when:

- Small datasets (<100K vectors) where exact results are required
- Search accuracy is critical and approximate results are not acceptable
- Baseline comparisons when evaluating approximate algorithms
- Simple use cases where setup simplicity is more important than performance

Performance characteristics:

- **Search accuracy:** 100% exact results (no approximation)
- **Search speed:** Linear time $O(n)$ - slower as dataset grows
- **Memory usage:** Minimal overhead, stores vectors as-is
- **Build time:** Fastest index construction (no preprocessing)

Trade-offs vs other algorithms:

- **vs HNSW:** Much slower search but exact results, faster index building
- **vs SVS-VAMANA:** Slower search and higher memory usage, but exact results

class FlatVectorField(*, name, type=FieldTypes.VECTOR, path=None, attrs)

Bases: BaseField

Vector field with FLAT (exact search) indexing for exact nearest neighbor search.

Create a new model by parsing and validating input data from keyword arguments.

Raises [ValidationError][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (*str*)
- **type** (*Literal[FieldTypes.VECTOR]*)
- **path** (*str | None*)
- **attrs** (*FlatVectorFieldAttributes*)

as_redis_field()

Convert schema field to Redis Field object

Return type

Field

attrs: *FlatVectorFieldAttributes*

Specified field attributes

```
model_config: ClassVar[ConfigDict] = {}
    Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].
```

```
type: Literal[FieldTypes.VECTOR]
    Field type
```

```
class FlatVectorFieldAttributes(*, dims, algorithm=VectorIndexAlgorithm.FLAT,
                               datatype=VectorDataType.FLOAT32,
                               distance_metric=VectorDistanceMetric.COSINE, initial_cap=None,
                               index_missing=False, block_size=None)
```

FLAT vector field attributes for exact nearest neighbor search.

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **dims** (*int*)
- **algorithm** (*Literal[VectorIndexAlgorithm.FLAT]*)
- **datatype** (*VectorDataType*)
- **distance_metric** (*VectorDistanceMetric*)
- **initial_cap** (*int | None*)
- **index_missing** (*bool*)
- **block_size** (*int | None*)

```
algorithm: Literal[VectorIndexAlgorithm.FLAT]
    The indexing algorithm (fixed as 'flat')
```

```
block_size: int | None
    Block size for processing (optional) - improves batch operation throughput
```

```
model_config: ClassVar[ConfigDict] = {}
    Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].
```

FLAT Example:

```
- name: embedding
  type: vector
  attrs:
    algorithm: flat
    dims: 768
    distance_metric: cosine
    datatype: float32
    # Optional: tune for batch processing
    block_size: 1024
```

Note: FLAT is recommended for small datasets or when exact results are mandatory. For larger datasets, consider HNSW or SVS-VAMANA for better performance.

SVS-VAMANA Configuration Utilities

For SVS-VAMANA indices, RedisVL provides utilities to help configure compression settings and estimate memory savings.

CompressionAdvisor

class CompressionAdvisor

Bases: object

Helper to recommend compression settings based on vector characteristics.

This class provides utilities to: - Recommend optimal SVS-VAMANA configurations based on vector dimensions and priorities - Estimate memory savings from compression and dimensionality reduction

Examples

```
>>> # Get recommendations for high-dimensional vectors
>>> config = CompressionAdvisor.recommend(dims=1536, priority="balanced")
>>> config.compression
'LeanVec4x8'
>>> config.reduce
768
```

```
>>> # Estimate memory savings
>>> savings = CompressionAdvisor.estimate_memory_savings(
...     compression="LeanVec4x8",
...     dims=1536,
...     reduce=768
... )
>>> savings
81.2
```

static estimate_memory_savings(*compression, dims, reduce=None*)

Estimate memory savings percentage from compression.

Calculates the percentage of memory saved compared to uncompressed float32 vectors.

Parameters

- **compression** (*str*) – Compression type (e.g., “LVQ4”, “LeanVec4x8”)
- **dims** (*int*) – Original vector dimensionality
- **reduce** (*int* | *None*) – Reduced dimensionality (for LeanVec compression)

Returns

Memory savings percentage (0-100)

Return type

float

Examples

```
>>> # LeanVec with dimensionality reduction
>>> CompressionAdvisor.estimate_memory_savings(
...     compression="LeanVec4x8",
...     dims=1536,
...     reduce=768
... )
81.2
```

```
>>> # LVQ without dimensionality reduction
>>> CompressionAdvisor.estimate_memory_savings(
...     compression="LVQ4",
...     dims=384
... )
87.5
```

static recommend(*dims*, *priority*='balanced', *datatype*=None)

Recommend compression settings based on dimensions and priorities.

Parameters

- **dims** (*int*) – Vector dimensionality (must be > 0)
- **priority** (*Literal['speed', 'memory', 'balanced']*) – Optimization priority: - “memory”: Maximize memory savings - “speed”: Optimize for query speed - “balanced”: Balance between memory and speed
- **datatype** (*str* / *None*) – Override datatype (default: float16 for high-dim, float32 for low-dim)

Returns

Complete SVS-VAMANA configuration including:

- algorithm: “svs-vamana”
- datatype: Recommended datatype
- compression: Compression type
- reduce: Dimensionality reduction (for LeanVec only)
- graph_max_degree: Graph connectivity
- construction_window_size: Build-time candidates
- search_window_size: Query-time candidates

Return type

dict

Raises

ValueError – If *dims* <= 0

Examples

```
>>> # High-dimensional embeddings (e.g., OpenAI ada-002)
>>> config = CompressionAdvisor.recommend(dims=1536, priority="memory")
>>> config.compression
'LeanVec4x8'
>>> config.reduce
768
```

```
>>> # Lower-dimensional embeddings
>>> config = CompressionAdvisor.recommend(dims=384, priority="speed")
>>> config.compression
'LVQ4x8'
```

SVSConfig

```
class SVSConfig(*, algorithm='svs-vamana', datatype=None, compression=None, reduce=None,
                 graph_max_degree=None, construction_window_size=None, search_window_size=None)
```

Bases: BaseModel

SVS-VAMANA configuration model.

Parameters

- **algorithm** (*Literal*['svs-vamana'])
- **datatype** (*str* | *None*)
- **compression** (*str* | *None*)
- **reduce** (*int* | *None*)
- **graph_max_degree** (*int* | *None*)
- **construction_window_size** (*int* | *None*)
- **search_window_size** (*int* | *None*)

algorithm

Always “svs-vamana”

Type

Literal['svs-vamana']

datatype

Vector datatype (float16, float32)

Type

str | *None*

compression

Compression type (LVQ4, LeanVec4x8, etc.)

Type

str | *None*

reduce

Reduced dimensionality (only for LeanVec)

Type

int | None

graph_max_degree

Max edges per node

Type

int | None

construction_window_size

Build-time candidates

Type

int | None

search_window_size

Query-time candidates

Type

int | None

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

Vector Algorithm Comparison

This section provides detailed guidance for choosing between vector search algorithms.

Algorithm Selection Guide

Table 1: Vector Algorithm Comparison

Algorithm	Best For	Performance	Memory Usage	Trade-offs
FLAT	Small datasets (<100K vectors)	100% recall, O(n) search	Minimal overhead	Exact but slow for large data
HNSW	General purpose (100K-1M+ vectors)	95-99% recall, O(log n) search	Moderate (graph overhead)	Fast approximate search
SVS-VAMANA	Large datasets with memory constraints	90-95% recall, O(log n) search	Low (with compression)	Intel-optimized, requires newer Redis

When to Use Each Algorithm

Choose FLAT when:

- Dataset size < 100,000 vectors
- Exact results are mandatory
- Simple setup is preferred
- Query latency is not critical

Choose HNSW when:

- Dataset size 100K - 1M+ vectors
- Need balanced speed and accuracy
- Cross-platform compatibility required
- Most common choice for production

Choose SVS-VAMANA when:

- Dataset size > 100K vectors
- Memory usage is a primary concern
- Running on Intel hardware
- Can accept 90-95% recall for memory savings

Performance Characteristics

Search Speed:

- FLAT: Linear time $O(n)$ - gets slower as data grows
- HNSW: Logarithmic time $O(\log n)$ - scales well
- SVS-VAMANA: Logarithmic time $O(\log n)$ - scales well

Memory Usage (1M vectors, 768 dims, float32):

- FLAT: ~3.1 GB (baseline)
- HNSW: ~3.7 GB (20% overhead for graph)
- SVS-VAMANA: 1.6-3.1 GB (depends on compression)

Recall Quality:

- FLAT: 100% (exact search)
- HNSW: 95-99% (tunable via `ef_runtime` at query time)
- SVS-VAMANA: 90-95% (tunable via `search_window_size` at query time, also depends on compression)

Migration Considerations

From FLAT to HNSW:

- Straightforward migration
- Expect slight recall reduction but major speed improvement
- Tune `ef_runtime` at query time to balance speed vs accuracy (no index rebuild needed)

From HNSW to SVS-VAMANA:

- Requires Redis $\geq$ 8.2 with Redis Search $\geq$ 2.8.10
- Change datatype to float16 or float32 if using others
- Consider compression options for memory savings

From SVS-VAMANA to others:

- May need to change datatype back if using float64/bfloat16
- HNSW provides similar performance with broader compatibility

For complete Redis field documentation, see: <https://redis.io/commands/ft.create/>

1.4.2 Search Index Classes

Class	Description
<i>SearchIndex</i>	Primary class to write, read, and search across data structures in Redis.
<i>AsyncSearchIndex</i>	Async version of the SearchIndex to write, read, and search across data structures in Redis.
<i>SearchResults</i>	List of result documents returned by a query, which also reports result completeness.

SearchIndex

```
class SearchIndex(schema, redis_client=None, redis_url=None, connection_kwargs=None,
                  validate_on_load=False, owns_client=None, **kwargs)
```

A search index class for interacting with Redis as a vector database.

The SearchIndex is instantiated with a reference to a Redis database and an IndexSchema (YAML path or dictionary object) that describes the various settings and field configurations.

```
from redisvl.index import SearchIndex

# initialize the index object with schema from file
index = SearchIndex.from_yaml(
    "schemas/schema.yaml",
    redis_url="redis://localhost:6379",
    validate_on_load=True
)

# create the index
index.create(overwrite=True, drop=False)
```

(continues on next page)

(continued from previous page)

```
# data is an iterable of dictionaries
index.load(data)

# delete index and data
index.delete(drop=True)
```

Pass `redis_client` to use a client you configured yourself. The index leaves such a client open when it is disconnected or garbage collected; pass `owns_client=True` to hand that responsibility over.

```
from redis import Redis
from redisvl.index import SearchIndex

client = Redis.from_url("redis://localhost:6379", socket_timeout=5)
index = SearchIndex.from_yaml("schemas/schema.yaml", redis_client=client)
```

Initialize the RedisVL search index with a schema and either a Redis client or a URL string with other connection kwargs.

Parameters

- **schema** (`IndexSchema`) – Index schema object.
- **redis_client** (`Optional[Redis]`) – An instantiated redis client.
- **redis_url** (`Optional[str]`) – The URL of the Redis server to connect to.
- **connection_kwargs** (`Dict[str, Any]`, *optional*) – Redis client connection args.
- **validate_on_load** (`bool`, *optional*) – Whether to validate data against schema when loading. Defaults to False.
- **owns_client** (`Optional[bool]`, *optional*) – Whether the index closes the Redis client when the index is disconnected or garbage collected. Defaults to None, meaning the index owns a client only if it created one itself. Pass True to hand over a client you created, or False to keep one the index would otherwise close, in which case closing it becomes your responsibility.

Raises

- **ValueError** – If schema is not an `IndexSchema`.
- **TypeError** – If `redis_client` is the wrong flavour, or if a keyword that no longer exists is passed. `connection_args` and `redis_kwargs` are both now `connection_kwargs`.

aggregate(*args, **kwargs)

Perform an aggregation operation against the index.

Wrapper around the aggregation API that adds the index name to the query and passes along the rest of the arguments to the `redis-py` `ft().aggregate()` method.

Returns

Raw Redis aggregation results.

Return type

Result

batch_query(queries, batch_size=10)

Execute a batch of queries and process results.

Parameters

- **queries** (*Sequence[BaseQuery]*)
- **batch_size** (*int*)

Return type

`list[list[dict[str, Any]]]`

batch_search(*queries, batch_size=10*)

Perform a search against the index for multiple queries.

This method takes a list of queries and optionally query params and returns a list of Result objects for each query. Results are returned in the same order as the queries.

NOTE: Cluster users may need to incorporate hash tags into their query to avoid cross-slot operations.

Parameters

- **queries** (*List[SearchParams]*) – The queries to search for.
- **batch_size** (*int, optional*) – The number of queries to search for at a time. Defaults to 10.

Returns

The search results for each query.

Return type

`List[Result]`

clear()

Clear all keys in Redis associated with the index, leaving the index available and in-place for future insertions or updates.

NOTE: This method requires custom behavior for Redis Cluster because here, we can't easily give control of the keys we're clearing to the user so they can separate them based on hash tag.

Note

The sweep enumerates through the index, so it removes only what the index currently returns, and it can stop early – against an index still being backfilled, or when a page's keys cannot be deleted. The returned count is the only signal, and 0 does not distinguish an empty index from a sweep that deleted nothing. Re-running is safe.

Returns

Count of records deleted from Redis.

Return type

`int`

create(*overwrite=False, drop=False*)

Create an index in Redis with the current schema and properties.

Parameters

- **overwrite** (*bool, optional*) – Whether to overwrite the index if it already exists. Defaults to False.
- **drop** (*bool, optional*) – Whether to drop all keys associated with the index in the case of overwriting. Defaults to False.

Raises

- **RuntimeError** – If the index already exists and ‘overwrite’ is False.
- **ValueError** – If no fields are defined for the index.

Return type

None

```
# create an index in Redis; only if one does not exist with given name
index.create()

# overwrite an index in Redis without dropping associated data
index.create(overwrite=True)

# overwrite an index in Redis; drop associated data (clean slate)
index.create(overwrite=True, drop=True)
```

delete(drop=True)

Delete the search index while optionally dropping all keys associated with the index.

FT.DROPINDEX resolves index aliases, so if this schema’s name happens to be an alias, the index the alias points at is dropped instead – along with its documents when drop is True. [exists\(\)](#) reports an alias as absent to keep [create\(\)](#) away from this path.

Parameters

drop (*bool*, *optional*) – Delete the key / documents pairs in the index. Defaults to True.

Raises

[RedisSearchError](#) – If the index cannot be deleted, for example when it does not exist. The original redis-py exception is available as `__cause__`.

disconnect()

Close the Redis client if this index owns it.

Always invalidates the cached SQL schema. When the index does not own the client (see `owns_client`), the client is left open and the index remains usable.

drop_by_filter(*filter_expression*, *, *batch_size*=500, *dry_run*=False, *allow_all*=False, *on_progress*=None)

Delete every document matching a filter expression.

Redis has no server-side “delete by query”, so RedisVL resolves the matching document keys and removes them with non-blocking UNLINK in batches. Matching documents leave the result set as they are deleted, so this re-queries from offset 0 each round (the same strategy as [clear\(\)](#)) and is not subject to the MAXSEARCHRESULTS limit (provided `batch_size` itself does not exceed it).

Parameters

- **filter_expression** (*Union[str, FilterExpression]*) – Selects the documents to delete. Prefer the escaping builders (`Tag`, `Num`, `Text`...) over raw filter strings; **never string-concatenate untrusted input into a filter** — unlike a read query, an injected predicate here deletes data.
- **batch_size** (*int*) – Number of documents to resolve and unlink per round-trip. Defaults to 500.
- **dry_run** (*bool*) – If True, report how many documents *would* be deleted (via a count query) without deleting anything.
- **allow_all** (*bool*) – Must be True to run against a match-all filter (`empty/"/"/None`). Prefer [clear\(\)](#) to intentionally empty the index.

- **on_progress** (*Optional[Callable[[int, int], None]]*) – Called after each batch with (processed, matched) — the cumulative documents deleted and the total matched at the start. Invoked synchronously (do not pass a coroutine); raising from it aborts the run (already-deleted documents stay deleted).

Returns

matched/processed counts, plus **completed** (False if the runaway backstop tripped) and **dry_run**. Read the fields explicitly (`result.processed`, `result.completed`).

Return type

BulkResult

 See also

[`update\_by\_filter\(\)`](#) (bulk partial update), [`drop\_documents\(\)`](#) / [`drop\_keys\(\)`](#) (delete by id/key), [`clear\(\)`](#) (delete all).

 Note

This operation is **not atomic** across the match set. Each key is unlinked atomically, but batches are applied incrementally with no rollback, so a crash or connection error mid-run leaves the already - deleted documents gone and the rest in place. Deletes are idempotent: re-running the same call after a failure removes only whatever still matches, converging on the intended state.

drop_documents(*ids*, *batch_size=500*)

Remove documents from the index by their document IDs.

This method converts document IDs to Redis keys automatically by applying the index's key prefix and separator configuration.

NOTE: Cluster users will need to incorporate hash tags into their document IDs and only call this method with documents from a single hash tag at a time.

Parameters

- **ids** (*Union[str, List[str]]*) – The document ID or IDs to remove from the index.
- **batch_size** (*int*) – Number of documents to delete per round-trip (standalone Redis only; cluster deletes in a single call after the shared-hash-tag check).

Returns

Count of documents deleted from Redis.

Return type

int

Raises

ValueError – On Redis Cluster, if the resolved keys do not all share a hash tag (a cross-slot DELETE is not permitted).

drop_keys(*keys*, *batch_size=500*)

Remove a specific entry or entries from the index by its key ID.

Uses UNLINK rather than DEL so memory reclamation runs on a background thread. This avoids blocking the main thread when a large number of keys are dropped at once (for example, scope-targeted SemanticCache invalidation). The returned count is unchanged.

Large key lists are unlinked in chunks of `batch_size`. On Redis Cluster, keys are unlinked individually so a chunk that spans hash slots does not raise `CROSSSLOT`.

Parameters

- **keys** (*Union[str, List[str]]*) – The document ID or IDs to remove from the index.
- **batch_size** (*int*) – Number of keys to unlink per round-trip.

Returns

Count of records deleted from Redis.

Return type

int

exists()

Check if the index exists in Redis.

Returns

True if the index exists, False otherwise.

Return type

bool

Raises

[*RedisSearchError*](#) – If the check fails for any reason other than the index not existing, such as a connection failure or insufficient permissions. The original `redis-py` exception is available as `__cause__`.

expire_keys(keys, ttl)

Set the expiration time for a specific entry or entries in Redis.

Parameters

- **keys** (*Union[str, List[str]]*) – The entry ID or IDs to set the expiration for.
- **ttl** (*int*) – The time-to-live in seconds.

Return type

int | list[int]

fetch(id)

Fetch an object from Redis by id.

The id is typically either a unique identifier, or derived from some domain-specific metadata combination (like a document id or chunk id).

Parameters

id (*str*) – The specified unique identifier for a particular document indexed in Redis.

Returns

The fetched object.

Return type

Dict[str, Any]

classmethod from_dict(schema_dict, **kwargs)

Create a SearchIndex from a dictionary.

Parameters

schema_dict (*Dict[str, Any]*) – A dictionary containing the schema.

Returns

A RedisVL SearchIndex object.

Return type*SearchIndex*

```
from redisvl.index import SearchIndex

index = SearchIndex.from_dict({
    "index": {
        "name": "my-index",
        "prefix": "rvl",
        "storage_type": "hash",
    },
    "fields": [
        {"name": "doc-id", "type": "tag"}
    ]
}, redis_url="redis://localhost:6379")
```

classmethod `from_existing(name, redis_client=None, redis_url=None, **kwargs)`

Initialize from an existing search index in Redis by index name.

Parameters

- **name** (*str*) – Name of the search index in Redis.
- **redis_client** (*Optional[Redis]*) – An instantiated redis client.
- **redis_url** (*Optional[str]*) – The URL of the Redis server to connect to.
- **owns_client** (*Optional[bool]*, *optional*) – Whether the index closes the client. Defaults to True when this method created the client from *redis_url*, and False when you supplied one.
- **connection_kwargs** (*Optional[Dict[str, Any]]*) – Redis client connection args, used only when this method creates the client from *redis_url*. Other keyword arguments are treated the same way.

Raises

ValueError – If *redis_url* or *redis_client* is not provided.

classmethod `from_yaml(schema_path, **kwargs)`

Create a *SearchIndex* from a YAML schema file.

Parameters

schema_path (*str*) – Path to the YAML schema file.

Returns

A RedisVL *SearchIndex* object.

Return type*SearchIndex*

```
from redisvl.index import SearchIndex

index = SearchIndex.from_yaml("schemas/schema.yaml", redis_url="redis://
↪localhost:6379")
```

info(*name=None*)

Get information about the index.

Parameters

name (*str*, *optional*) – Index name to fetch info about. Defaults to None.

Returns

A dictionary containing the information about the index.

Return type

dict

invalidate_sql_schema_cache()

Clear cached sql-redis executors and schema state for this index.

Return type

None

key(id)

Construct a redis key as a combination of an index key prefix (optional) and specified id.

The id is typically either a unique identifier, or derived from some domain-specific metadata combination (like a document id or chunk id).

Parameters

id (*str*) – The specified unique identifier for a particular document indexed in Redis.

Returns

The full Redis key including key prefix and value as a string.

Return type

str

listall()

List all search indices in Redis database.

Returns

The list of indices in the database.

Return type

List[str]

load(data, id_field=None, keys=None, ttl=None, preprocess=None, batch_size=None)

Load objects to the Redis database. Returns the list of keys loaded to Redis.

RedisVL automatically handles constructing the object keys, batching, optional preprocessing steps, and setting optional expiration (TTL policies) on keys.

Parameters

- **data** (*Iterable[Any]*) – An iterable of objects to store.
- **id_field** (*Optional[str], optional*) – Specified field used as the id portion of the redis key (after the prefix) for each object. Defaults to None.
- **keys** (*Optional[Iterable[str]], optional*) – Optional iterable of keys. Must match the length of objects if provided. Defaults to None.
- **ttl** (*Optional[int], optional*) – Time-to-live in seconds for each key. Defaults to None.
- **preprocess** (*Optional[Callable], optional*) – A function to preprocess objects before storage. Defaults to None.
- **batch_size** (*Optional[int], optional*) – Number of objects to write in a single Redis pipeline execution. Defaults to class's default batch size.

Returns

List of keys loaded to Redis.

Return type

List[str]

Raises

- ***SchemaValidationError*** – If validation fails when `validate_on_load` is enabled.
- ***RedisVLError*** – If there's an error loading data to Redis.

paginate(*query*, *page_size=30*)

Execute a given query against the index and return results in paginated batches.

This method accepts a RedisVL query instance, enabling pagination of results which allows for subsequent processing over each batch with a generator.

Parameters

- **query** (*BaseQuery*) – The search query to be executed.
- **page_size** (*int*, *optional*) – The number of results to return in each batch. Defaults to 30.

Yields

A generator yielding non-empty `SearchResults` batches.

Raises

- **TypeError** – If the `page_size` argument is not of type `int`.
- **ValueError** – If the `page_size` argument is less than or equal to zero.

Return type*Generator*

```
# Iterate over paginated search results in batches of 10
for result_batch in index.paginate(query, page_size=10):
    # Process each batch of results
    pass
```

Note

The `page_size` parameter controls the number of items each result batch contains. Adjust this value based on performance considerations and the expected volume of search results.

Note

For stable pagination, the query must have a `sort_by` clause on a **unique** field. Redis documents that `LIMIT` without sorting is non-deterministic, so pages may otherwise repeat or miss documents. Very deep pagination is also bounded server-side by `search-max-search-results` (1,000,000 by default, but 10,000 on some managed tiers), past which the search errors rather than ending.

Note

A yielded batch may contain fewer than `page_size` documents, and an empty batch is never yielded. On Redis 8+ a matched document whose data expires while the search is running is skipped; each batch

reports how many were skipped via `dropped_count` (and `complete`), including skips inherited from a page that was dropped in its entirety. Pagination still runs to the end of the result set in that case.

query(*query*)

Execute a query on the index.

This method takes a `BaseQuery`, `AggregationQuery`, or `HybridQuery` object directly, and handles post-processing of the search.

Parameters

query (`Union[BaseQuery, AggregationQuery, HybridQuery]`) – The query to run.

Returns

A list of search results.

Return type

`List[Result]`

```
from redisvl.query import VectorQuery

query = VectorQuery(
    vector=[0.16, -0.34, 0.98, 0.23],
    vector_field_name="embedding",
    num_results=3
)

results = index.query(query)
```

search(*args, **kwargs)

Perform a search against the index.

Wrapper around the search API that adds the index name to the query and passes along the rest of the arguments to the `redis-py` `ft().search()` method.

Returns

Raw Redis search results.

Return type

`Result`

update_by_filter(*filter_expression*, *values*, *, *batch_size*=500, *dry_run*=False, *allow_all*=False, *on_progress*=None)

Set values on every document matching a filter expression.

This is a partial update: fields not present in `values` are left untouched. For hash indexes the fields are written with `HSET`; for JSON indexes they are merged at the document root (\$) with `JSON.MERGE` (RFC 7396), so nested objects merge recursively, arrays are replaced wholesale, and a `None` value deletes that path.

The read phase (an `FT.AGGREGATE` cursor) is drained in full *before* any write, holding every matching key in memory in between. Interleaving the two does not terminate: `Redisearch` has no in-place update, so each write appends a new, higher document id ahead of the cursor, which hands that document back to be written again. Memory use is therefore proportional to the match count; for very large match sets, narrow the filter and run in partitions (see “Bulk Delete and Update” in the search-and-indexing concepts guide).

Parameters

- **filter_expression** (*Union[str, FilterExpression]*) – Selects the documents to update. Prefer the escaping builders (`Tag`, `Num...`) over raw filter strings; **never string-concatenate untrusted input into a filter** — an injected predicate here mutates data.
- **values** (*Dict[str, Any]*) – Field/value pairs to set on each match. Values are written **as-is with no schema validation** (unlike `load()`): callers must pre-encode vectors/bytes and format numerics as the schema expects. For JSON indexes, keys must match the document's JSON **layout**, not the schema field name — a field indexed at a nested path (e.g. `$.metadata.status`) must be passed nested (`{"metadata": {"status": ...}}`); passing the flat field name writes the wrong path and leaves the indexed field unchanged. Only static values are supported (no callable/expression transforms).
- **batch_size** (*int*) – Number of documents to update per round-trip.
- **dry_run** (*bool*) – If True, report how many documents *would* be updated without writing anything.
- **allow_all** (*bool*) – Must be True to run against a match-all filter.
- **on_progress** (*Optional[Callable[[int, int], None]]*) – Called after each write batch with (`processed`, `matched`). Invoked synchronously (do not pass a coroutine); raising from it aborts the run.

Returns

`matched/processed` counts, plus `completed` (always True for update — it runs to completion or raises) and `dry_run`.

Return type

`BulkResult`

 See also

`drop_by_filter()`, `load()` (validated whole-document upsert by key).

 Note

This operation is **not atomic** across the match set. Each document is updated atomically (one `HSET/JSON.MERGE`), but batches use a non-transactional pipeline and are applied incrementally with no rollback, so a crash or connection error mid-run can leave some documents updated and others not. Because the update is a fixed field set, it is idempotent: re-running the same call after a failure converges on the intended state.

Keys are resolved before writing, so a document may be deleted by another client in between. Each write is conditional on the key still existing (applied atomically), so such a document is **skipped rather than recreated** as a partial document; it simply isn't counted in `processed`. `processed` therefore reflects the documents actually written, which can be less than `matched` under concurrent deletion.

`processed` counts **distinct** documents — a document is written *at most* once, even though the resolving cursor can emit it on more than one page — but there is no *at least* once guarantee. It can exceed `matched` if the match set grew after the initial count, and fall short of it either from a concurrent deletion or because the cursor itself returned fewer rows than `matched` (which Redis documents as possible). A short `processed` is therefore not evidence of any particular cause.

property client: `Redis` | `RedisCluster` | `None`

The underlying redis-py client object.

property key_separator: str

The optional separator between a defined prefix and key value in forming a Redis key.

property name: str

The name of the Redis search index.

property prefix: str

The key prefix used in forming Redis keys.

For multi-prefix indexes, returns the first prefix.

property prefixes: list[str]

All key prefixes configured for this index.

property storage_type: StorageType

The underlying storage type for the search index; either hash or json.

AsyncSearchIndex

```
class AsyncSearchIndex(schema, *, redis_url=None, redis_client=None, connection_kwargs=None,
                        validate_on_load=False, owns_client=None, **kwargs)
```

A search index class for interacting with Redis as a vector database in async-mode.

The AsyncSearchIndex is instantiated with a reference to a Redis database and an IndexSchema (YAML path or dictionary object) that describes the various settings and field configurations.

```
from redisvl.index import AsyncSearchIndex

# initialize the index object with schema from file
index = AsyncSearchIndex.from_yaml(
    "schemas/schema.yaml",
    redis_url="redis://localhost:6379",
    validate_on_load=True
)

# create the index
await index.create(overwrite=True, drop=False)

# data is an iterable of dictionaries
await index.load(data)

# delete index and data
await index.delete(drop=True)
```

Pass `redis_client` to use a client you configured yourself. The index leaves such a client open when it is disconnected or garbage collected; pass `owns_client=True` to hand that responsibility over.

```
from redis.asyncio import Redis
from redisvl.index import AsyncSearchIndex

client = Redis.from_url("redis://localhost:6379", socket_timeout=5)
index = AsyncSearchIndex.from_yaml("schemas/schema.yaml", redis_client=client)
```

Initialize the RedisVL async search index with a schema.

Parameters

- **schema** ([IndexSchema](#)) – Index schema object.
- **redis_url** (*Optional[str], optional*) – The URL of the Redis server to connect to.
- **redis_client** (*Optional[AsyncRedis]*) – An instantiated redis client.
- **connection_kwargs** (*Optional[Dict[str, Any]]*) – Redis client connection args.
- **validate_on_load** (*bool, optional*) – Whether to validate data against schema when loading. Defaults to False.
- **owns_client** (*Optional[bool], optional*) – Whether the index closes the Redis client when the index is disconnected or garbage collected. Defaults to None, meaning the index owns a client only if it created one itself. Pass True to hand over a client you created, or False to keep one the index would otherwise close, in which case closing it becomes your responsibility.

Raises

- **ValueError** – If schema is not an IndexSchema.
- **TypeError** – If redis_client is the wrong flavour, or if a keyword that no longer exists is passed. connection_args and redis_kwargs are both now connection_kwargs.

async aggregate(*args, **kwargs)

Perform an aggregation operation against the index.

Wrapper around the aggregation API that adds the index name to the query and passes along the rest of the arguments to the redis-py ft().aggregate() method.

Returns

Raw Redis aggregation results.

Return type

Result

async batch_query(queries, batch_size=10)

Asynchronously execute a batch of queries and process results.

Parameters

- **queries** (*list[BaseQuery]*)
- **batch_size** (*int*)

Return type

list[list[dict[str, Any]]]

async batch_search(queries, batch_size=10)

Asynchronously execute a batch of search queries.

This method takes a list of search queries and executes them in batches to improve performance when dealing with multiple queries.

NOTE: Cluster users may need to incorporate hash tags into their query to avoid cross-slot operations.

Parameters

- **queries** (*List[SearchParams]*) – A list of search queries to execute. Each query can be either a string or a tuple of (query, params).
- **batch_size** (*int, optional*) – The number of queries to execute in each batch. Defaults to 10.

Returns

A list of search results corresponding to each query.

Return type

List[Result]

```
queries = [
    "hello world",
    ("goodbye world", {"num_results": 5}),
]

results = await index.batch_search(queries)
```

async clear()

Clear all keys in Redis associated with the index, leaving the index available and in-place for future insertions or updates.

NOTE: This method requires custom behavior for Redis Cluster because here, we can't easily give control of the keys we're clearing to the user so they can separate them based on hash tag.

See [SearchIndex.clear\(\)](#) for the sweep's caveats, which apply identically here.

Returns

Count of records deleted from Redis.

Return type

int

async create(overwrite=False, drop=False)

Asynchronously create an index in Redis with the current schema and properties.

Parameters

- **overwrite** (*bool*, *optional*) – Whether to overwrite the index if it already exists. Defaults to False.
- **drop** (*bool*, *optional*) – Whether to drop all keys associated with the index in the case of overwriting. Defaults to False.

Raises

- **RuntimeError** – If the index already exists and 'overwrite' is False.
- **ValueError** – If no fields are defined for the index.

Return type

None

```
# create an index in Redis; only if one does not exist with given name
await index.create()

# overwrite an index in Redis without dropping associated data
await index.create(overwrite=True)

# overwrite an index in Redis; drop associated data (clean slate)
await index.create(overwrite=True, drop=True)
```

async delete(drop=True)

Delete the search index.

FT.DROPINDEX resolves index aliases, so if this schema's name happens to be an alias, the index the alias points at is dropped instead – along with its documents when `drop` is `True`. `exists()` reports an alias as absent to keep `create()` away from this path.

Parameters

drop (*bool*, *optional*) – Delete the documents in the index. Defaults to `True`.

Raises

RedisSearchError – If the index cannot be deleted, for example when it does not exist. The original `redis-py` exception is available as `__cause__`.

async disconnect()

Close the Redis client if this index owns it.

Always invalidates the cached SQL schema. When the index does not own the client (see `owns_client`), the client is left open and the index remains usable.

disconnect_sync()

Close an owned Redis client from synchronous code.

For callers outside an event loop, such as `__del__` or a shutdown hook. Honours `owns_client` exactly as `disconnect()` does.

Does nothing if a loop is already running, since it cannot await the close from inside one. Use `disconnect()` there.

async drop_by_filter(filter_expression, *, batch_size=500, dry_run=False, allow_all=False, on_progress=None)

Delete every document matching a filter expression (async).

See `SearchIndex.drop_by_filter()` for full semantics.

Parameters

- **filter_expression** (*str* | `FilterExpression`)
- **batch_size** (*int*)
- **dry_run** (*bool*)
- **allow_all** (*bool*)
- **on_progress** (`Callable[[int, int], None]` | `None`)

Return type

BulkResult

async drop_documents(ids, batch_size=500)

Remove documents from the index by their document IDs.

This method converts document IDs to Redis keys automatically by applying the index's key prefix and separator configuration.

NOTE: Cluster users will need to incorporate hash tags into their document IDs and only call this method with documents from a single hash tag at a time.

Parameters

- **ids** (`Union[str, List[str]]`) – The document ID or IDs to remove from the index.

- **batch_size** (*int*) – Number of documents to delete per round-trip (standalone Redis only; cluster deletes in a single call after the shared-hash-tag check).

Returns

Count of documents deleted from Redis.

Return type

int

Raises

ValueError – On Redis Cluster, if the resolved keys do not all share a hash tag (a cross-slot DELETE is not permitted).

async drop_keys(*keys*, *batch_size=500*)

Remove a specific entry or entries from the index by it's key ID.

Uses UNLINK rather than DEL so memory reclamation runs on a background thread. This avoids blocking the main thread when a large number of keys are dropped at once (for example, scope-targeted `SemanticCache` invalidation). The returned count is unchanged.

Large key lists are unlinked in chunks of *batch_size*. On Redis Cluster, keys are unlinked individually so a chunk that spans hash slots does not raise `CROSSSLOT`.

Parameters

- **keys** (*Union[str, List[str]]*) – The document ID or IDs to remove from the index.
- **batch_size** (*int*) – Number of keys to unlink per round-trip.

Returns

Count of records deleted from Redis.

Return type

int

async exists()

Check if the index exists in Redis.

Returns

True if the index exists, False otherwise.

Return type

bool

Raises

RedisSearchError – If the check fails for any reason other than the index not existing, such as a connection failure or insufficient permissions. The original `redis-py` exception is available as `__cause__`.

async expire_keys(*keys*, *tvl*)

Set the expiration time for a specific entry or entries in Redis.

Parameters

- **keys** (*Union[str, List[str]]*) – The entry ID or IDs to set the expiration for.
- **tvl** (*int*) – The time-to-live in seconds.

Return type

int | list[int]

async fetch(id)

Asynchronously etch an object from Redis by id. The id is typically either a unique identifier, or derived from some domain-specific metadata combination (like a document id or chunk id).

Parameters

id (*str*) – The specified unique identifier for a particular document indexed in Redis.

Returns

The fetched object.

Return type

Dict[str, Any]

classmethod from_dict(schema_dict, **kwargs)

Create a SearchIndex from a dictionary.

Parameters

schema_dict (*Dict[str, Any]*) – A dictionary containing the schema.

Returns

A RedisVL SearchIndex object.

Return type

SearchIndex

```
from redisvl.index import SearchIndex

index = SearchIndex.from_dict({
    "index": {
        "name": "my-index",
        "prefix": "rvl",
        "storage_type": "hash",
    },
    "fields": [
        {"name": "doc-id", "type": "tag"}
    ]
}, redis_url="redis://localhost:6379")
```

async classmethod from_existing(name, redis_client=None, redis_url=None, **kwargs)

Initialize from an existing search index in Redis by index name.

Parameters

- **name** (*str*) – Name of the search index in Redis.
- **redis_client** (*Optional[Redis]*) – An instantiated redis client.
- **redis_url** (*Optional[str]*) – The URL of the Redis server to connect to.
- **owns_client** (*Optional[bool]*, *optional*) – Whether the index closes the client. Defaults to True when this method created the client from *redis_url*, and False when you supplied one.
- **connection_kwargs** (*Optional[Dict[str, Any]]*) – Redis client connection args, used only when this method creates the client from *redis_url*. Other keyword arguments are treated the same way.

classmethod from_yaml(schema_path, **kwargs)

Create a SearchIndex from a YAML schema file.

Parameters

schema_path (*str*) – Path to the YAML schema file.

Returns

A RedisVL SearchIndex object.

Return type

SearchIndex

```
from redisvl.index import SearchIndex

index = SearchIndex.from_yaml("schemas/schema.yaml", redis_url="redis://
↪localhost:6379")
```

async info(*name=None*)

Get information about the index.

Parameters

name (*str*, *optional*) – Index name to fetch info about. Defaults to None.

Returns

A dictionary containing the information about the index.

Return type

dict

invalidate_sql_schema_cache()

Clear cached sql-redis executors and schema state for this index.

Return type

None

key(*id*)

Construct a redis key as a combination of an index key prefix (optional) and specified id.

The id is typically either a unique identifier, or derived from some domain-specific metadata combination (like a document id or chunk id).

Parameters

id (*str*) – The specified unique identifier for a particular document indexed in Redis.

Returns

The full Redis key including key prefix and value as a string.

Return type

str

async listall()

List all search indices in Redis database.

Returns

The list of indices in the database.

Return type

List[str]

load(*data, id_field=None, keys=None, ttl=None, preprocess=None, concurrency=None, batch_size=None*)

Asynchronously load objects to Redis. Returns the list of keys loaded to Redis.

RedisVL automatically handles constructing the object keys, batching, optional preprocessing steps, and setting optional expiration (TTL policies) on keys.

Parameters

- **data** (*Iterable[Any]*) – An iterable of objects to store.
- **id_field** (*Optional[str]*, *optional*) – Specified field used as the id portion of the redis key (after the prefix) for each object. Defaults to None.
- **keys** (*Optional[Iterable[str]]*, *optional*) – Optional iterable of keys. Must match the length of objects if provided. Defaults to None.
- **ttl** (*Optional[int]*, *optional*) – Time-to-live in seconds for each key. Defaults to None.
- **preprocess** (*Optional[Callable]*, *optional*) – A function to preprocess objects before storage. Defaults to None.
- **batch_size** (*Optional[int]*, *optional*) – Number of objects to write in a single Redis pipeline execution. Defaults to class's default batch size.
- **concurrency** (*int* | *None*)

Returns

List of keys loaded to Redis.

Return type

List[str]

Raises

- **SchemaValidationError** – If validation fails when `validate_on_load` is enabled.
- **RedisVLError** – If there's an error loading data to Redis.

```
data = [{"test": "foo"}, {"test": "bar"}]

# simple case
keys = await index.load(data)

# set 360 second ttl policy on data
keys = await index.load(data, ttl=360)

# load data with predefined keys
keys = await index.load(data, keys=["rvl:foo", "rvl:bar"])

# load data with preprocessing step
def add_field(d):
    d["new_field"] = 123
    return d
keys = await index.load(data, preprocess=add_field)
```

async paginate(query, page_size=30)

Execute a given query against the index and return results in paginated batches.

This method accepts a RedisVL query instance, enabling async pagination of results which allows for subsequent processing over each batch with a generator.

Parameters

- **query** (*BaseQuery*) – The search query to be executed.
- **page_size** (*int*, *optional*) – The number of results to return in each batch. Defaults to 30.

Yields

An async generator yielding non-empty `SearchResults` batches.

Raises

- **TypeError** – If the `page_size` argument is not of type `int`.
- **ValueError** – If the `page_size` argument is less than or equal to zero.

Return type

AsyncGenerator

```
# Iterate over paginated search results in batches of 10
async for result_batch in index.paginate(query, page_size=10):
    # Process each batch of results
    pass
```

Note

The `page_size` parameter controls the number of items each result batch contains. Adjust this value based on performance considerations and the expected volume of search results.

Note

For stable pagination, the query must have a `sort_by` clause on a **unique** field. Redis documents that `LIMIT` without sorting is non-deterministic, so pages may otherwise repeat or miss documents. Very deep pagination is also bounded server-side by `search-max-search-results` (1,000,000 by default, but 10,000 on some managed tiers), past which the search errors rather than ending.

Note

A yielded batch may contain fewer than `page_size` documents, and an empty batch is never yielded. On Redis 8+ a matched document whose data expires while the search is running is skipped; each batch reports how many were skipped via `dropped_count` (and `complete`), including skips inherited from a page that was dropped in its entirety. Pagination still runs to the end of the result set in that case.

async query(query)

Asynchronously execute a query on the index.

This method takes a `BaseQuery`, `AggregationQuery`, `HybridQuery`, or `SQLQuery` object directly, runs the search, and handles post-processing of the search.

Parameters

query (*Union[BaseQuery, AggregationQuery, HybridQuery, SQLQuery]*) – The query to run.

Returns

A list of search results.

Return type

`List[Result]`

```
from redisvl.query import VectorQuery

query = VectorQuery(
    vector=[0.16, -0.34, 0.98, 0.23],
    vector_field_name="embedding",
    num_results=3
)

results = await index.query(query)
```

async search(*args, **kwargs)

Perform an async search against the index.

Wrapper around the search API that adds the index name to the query and passes along the rest of the arguments to the redis-py `ft().search()` method.

Returns

Raw Redis search results.

Return type

Result

async update_by_filter(filter_expression, values, *, batch_size=500, dry_run=False, allow_all=False, on_progress=None)

Set values on every document matching a filter expression (async).

See [SearchIndex.update_by_filter\(\)](#) for full semantics.

Parameters

- **filter_expression** (*str* | *FilterExpression*)
- **values** (*dict[str, Any]*)
- **batch_size** (*int*)
- **dry_run** (*bool*)
- **allow_all** (*bool*)
- **on_progress** (*Callable[[int, int], None]* | *None*)

Return type

BulkResult

property client: *Redis* | *RedisCluster* | *None*

The underlying redis-py client object.

property key_separator: *str*

The optional separator between a defined prefix and key value in forming a Redis key.

property name: *str*

The name of the Redis search index.

property prefix: *str*

The key prefix used in forming Redis keys.

For multi-prefix indexes, returns the first prefix.

property prefixes: *list[str]*

All key prefixes configured for this index.

property storage_type: `StorageType`

The underlying storage type for the search index; either hash or json.

SearchResults

class SearchResults(*iterable=()*, *dropped_count=0*)

A list of result documents that also reports result completeness.

This is a drop-in `list` — iteration, indexing, `len()`, and every other list operation behave exactly as before, so existing callers need no changes. It additionally carries:

- **dropped_count**: the number of matched documents that were skipped because their field payload came back missing (the Redis 8.8+ background-search TTL/expiry race); `0` in the normal case.
- **complete**: `False` when `dropped_count > 0`, i.e. the result set may be missing one or more matches that the server reported but could not materialize.

Callers that need completeness guarantees — audit/compliance queries, eval harnesses, or agents making control-flow decisions — can inspect these to detect and react to a partial result (e.g. retry, reconcile against a `CountQuery`, or annotate a downstream answer as incomplete). Everyone else can ignore them. Note that operations returning a new list (slicing, `sorted()`, concatenation) yield a plain `list` without this metadata.

Parameters

- **iterable** (*Any*)
- **dropped_count** (*int*)

property complete: `bool`

True when no matched documents were dropped from this result set.

1.4.3 Vector

The `Vector` class in RedisVL is a container that encapsulates a numerical vector, its datatype, corresponding index field name, and optional importance weight. It is used when constructing multi-vector queries using the `MultiVectorQuery` class.

Vector

class Vector(***, *vector*, *field_name*, *dtype='float32'*, *weight=1.0*, *max_distance=2.0*)

Simple object containing the necessary arguments to perform a multi vector query.

Args: `vector`: The vector values as a list of floats or bytes `field_name`: The name of the vector field to search `dtype`: The data type of the vector (default: “float32”) `weight`: The weight for this vector in the combined score (default: 1.0) `max_distance`: The maximum distance for vector range search (default: 2.0, range: [0.0, 2.0])

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **vector** (*list[float] | bytes*)
- **field_name** (*str*)

- **dtype** (*str*)
- **weight** (*float*)
- **max_distance** (*float*)

validate_vector()

If the vector passed in is an array of float convert it to a byte string.

Return type

Self

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

1.4.4 Query

Query classes in RedisVL provide a structured way to define simple or complex queries for different use cases. Each query class wraps the `redis-py` Query module <https://github.com/redis/redis-py/blob/master/redis/commands/search/query.py> with extended functionality for ease-of-use.

VectorQuery

```
class VectorQuery(vector, vector_field_name, return_fields=None, filter_expression=None, dtype='float32',
                  num_results=10, return_score=True, dialect=2, sort_by=None, in_order=False,
                  hybrid_policy=None, batch_size=None, ef_runtime=None, search_window_size=None,
                  use_search_history=None, search_buffer_capacity=None,
                  normalize_vector_distance=False)
```

Bases: `BaseVectorQuery`, `BaseQuery`

A query for running a vector search along with an optional filter expression.

Parameters

- **vector** (*List[float]*) – The vector to perform the vector search with.
- **vector_field_name** (*str*) – The name of the vector field to search against in the database.
- **return_fields** (*List[str]*) – The declared fields to return with search results.
- **filter_expression** (*Union[str, FilterExpression]*, *optional*) – A filter to apply along with the vector search. Defaults to `None`.
- **dtype** (*str*, *optional*) – The dtype of the vector. Defaults to “float32”.
- **num_results** (*int*, *optional*) – The top k results to return from the vector search. Defaults to 10.
- **return_score** (*bool*, *optional*) – Whether to return the vector distance. Defaults to `True`.
- **dialect** (*int*, *optional*) – The Redis Search query dialect. Defaults to 2.
- **sort_by** (*Optional[SortSpec]*) – The field(s) to order the results by. Can be: - *str*: single field name - *Tuple[str, str]*: (field_name, “ASC”|”DESC”) - *List*: list of fields or tuples Note: Only the first field is used for Redis sorting. Defaults to `None`. Results will be ordered by vector distance.

- **in_order** (*bool*) – Requires the terms in the field to have the same order as the terms in the query filter, regardless of the offsets between them. Defaults to False.
- **hybrid_policy** (*Optional[str]*) – Controls how filters are applied during vector search. Options are “BATCHES” (paginates through small batches of nearest neighbors) or “ADHOC_BF” (computes scores for all vectors passing the filter). “BATCHES” mode is typically faster for queries with selective filters. “ADHOC_BF” mode is better when filters match a large portion of the dataset. Defaults to None, which lets Redis auto-select the optimal policy.
- **batch_size** (*Optional[int]*) – When hybrid_policy is “BATCHES”, controls the number of vectors to fetch in each batch. Larger values may improve performance at the cost of memory usage. Only applies when hybrid_policy=“BATCHES”. Defaults to None, which lets Redis auto-select an appropriate batch size.
- **ef_runtime** (*Optional[int]*) – Controls the size of the dynamic candidate list for HNSW algorithm at query time. Higher values improve recall at the expense of slower search performance. Defaults to None, which uses the index-defined value.
- **search_window_size** (*Optional[int]*) – The size of the search window for SVS-VAMANA KNN searches. Increasing this value generally yields more accurate but slower search results. Defaults to None, which uses the index-defined value (typically 10).
- **use_search_history** (*Optional[str]*) – For SVS-VAMANA indexes, controls whether to use the search buffer or entire search history. Options are “OFF”, “ON”, or “AUTO”. “AUTO” is always evaluated internally as “ON”. Using the entire history may yield a slightly better graph at the cost of more search time. Defaults to None, which uses the index-defined value (typically “AUTO”).
- **search_buffer_capacity** (*Optional[int]*) – Tuning parameter for SVS-VAMANA indexes using two-level compression (LVQ<X>x<Y> or LeanVec types). Determines the number of vector candidates to collect in the first level of search before the re-ranking level. Defaults to None, which uses the index-defined value (typically SEARCH_WINDOW_SIZE).
- **normalize_vector_distance** (*bool*) – Redis supports 3 distance metrics: L2 (euclidean), IP (inner product), and COSINE. By default, L2 distance returns an unbounded value. COSINE distance returns a value between 0 and 2. IP returns a value determined by the magnitude of the vector. Setting this flag to true converts COSINE and L2 distance to a similarity score between 0 and 1. Note: setting this flag to true for IP will throw a warning since by definition COSINE similarity is normalized IP.

Raises

TypeError – If filter_expression is not of type redisvl.query.FilterExpression

Note

Learn more about vector queries in Redis: <https://redis.io/docs/latest/develop/ai/search-and-query/vectors/#knn-vector-search>

dialect (*dialect*)

Add a dialect field to the query.

- **dialect** - dialect version to execute the query under

Parameters

dialect (*int*)

Return type

Query

expander(*expander*)

Add an expander field to the query.

- **expander** - the name of the expander

Parameters

expander (*str*)

Return type

Query

in_order()

Match only documents where the query terms appear in the same order in the document. i.e., for the query “hello world”, we do not match “world hello”

Return type

Query

language(*language*)

Analyze the query as being in the specified language.

Parameters

language (*str*) – The language (e.g. *chinese* or *english*)

Return type

Query

limit_fields(**fields*)

Limit the search to specific TEXT fields only.

- **fields**: Each element should be a string, case sensitive field name from the defined schema.

Parameters

fields (*str*)

Return type

Query

limit_ids(**ids*)

Limit the results to a specific set of pre-known document ids of any length.

Return type

Query

no_content()

Set the query to only return ids and not the document content.

Return type

Query

no_stopwords()

Prevent the query from being filtered for stopwords. Only useful in very big queries that you are certain contain no stopwords.

Return type

Query

paging(*offset, num*)

Set the paging for the query (defaults to 0..10).

- **offset**: Paging offset for the results. Defaults to 0
- **num**: How many results do we want

Parameters

- **offset** (*int*)
- **num** (*int*)

Return type

Query

query_string()

Return the query string of this query only.

Return type

str

return_fields(**fields, skip_decode=None*)

Set the fields to return with search results.

Parameters

- ***fields** – Variable number of field names to return.
- **skip_decode** (*str | list[str] | None*) – Optional field name or list of field names that should not be decoded. Useful for binary data like embeddings.

Returns

Returns the query object for method chaining.

Return type

self

Raises

TypeError – If skip_decode is not a string, list, or None.

scorer(*scorer*)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Since Redis 8.0 default was changed to BM25STD.

Parameters

scorer (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)

Return type

Query

set_batch_size(*batch_size*)

Set the batch size for the query.

Parameters

batch_size (*int*) – The batch size to use when hybrid_policy is “BATCHES”.

Raises

- **TypeError** – If batch_size is not an integer
- **ValueError** – If batch_size is not positive

set_ef_runtime(*ef_runtime*)

Set the EF_RUNTIME parameter for the query.

Parameters

ef_runtime (*int*) – The EF_RUNTIME value to use for HNSW algorithm. Higher values improve recall at the expense of slower search.

Raises

- **TypeError** – If *ef_runtime* is not an integer
- **ValueError** – If *ef_runtime* is not positive

set_filter(*filter_expression=None*)

Set the filter expression for the query.

Parameters

filter_expression (*Optional[Union[str, FilterExpression]], optional*) – The filter expression or query string to use on the query.

Raises

TypeError – If *filter_expression* is not a valid `FilterExpression` or string.

set_hybrid_policy(*hybrid_policy*)

Set the hybrid policy for the query.

Parameters

hybrid_policy (*str*) – The hybrid policy to use. Options are “BATCHES” or “AD-HOC_BF”.

Raises

ValueError – If *hybrid_policy* is not one of the valid options

set_search_buffer_capacity(*search_buffer_capacity*)

Set the SEARCH_BUFFER_CAPACITY parameter for the query.

Parameters

search_buffer_capacity (*int*) – Tuning parameter for SVS-VAMANA indexes using two-level compression. Determines the number of vector candidates to collect in the first level of search before the re-ranking level.

Raises

- **TypeError** – If *search_buffer_capacity* is not an integer
- **ValueError** – If *search_buffer_capacity* is not positive

set_search_window_size(*search_window_size*)

Set the SEARCH_WINDOW_SIZE parameter for the query.

Parameters

search_window_size (*int*) – The size of the search window for SVS-VAMANA KNN searches. Increasing this value generally yields more accurate but slower search results.

Raises

- **TypeError** – If *search_window_size* is not an integer
- **ValueError** – If *search_window_size* is not positive

set_use_search_history(*use_search_history*)

Set the USE_SEARCH_HISTORY parameter for the query.

Parameters

use_search_history (*str*) – For SVS-VAMANA indexes, controls whether to use the search buffer or entire search history. Options are “OFF”, “ON”, or “AUTO”.

Raises

- **TypeError** – If `use_search_history` is not a string
- **ValueError** – If `use_search_history` is not one of “OFF”, “ON”, or “AUTO”

slop(*slop*)

Allow a maximum of N intervening non-matched terms between phrase terms (0 means exact phrase).

Parameters

slop (*int*)

Return type

Query

sort_by(*sort_spec=None, asc=True*)

Set the sort order for query results.

This method supports sorting by single or multiple fields. Note that Redis Search natively supports only a single SORTBY field. When multiple fields are specified, only the FIRST field is used for the Redis SORTBY clause.

Parameters

- **sort_spec** (*str | tuple[str, str] | list[str | tuple[str, str]] | None*) – Sort specification in various formats: - `str`: single field name - `Tuple[str, str]`: (`field_name`, “ASC”|“DESC”) - `List`: list of field names or tuples
- **asc** (*bool*) – Default sort direction when not specified (only used when `sort_spec` is a string). Defaults to `True` (ascending).

Returns

Returns the query object for method chaining.

Return type

`self`

Raises

- **TypeError** – If `sort_spec` is not a valid type.
- **ValueError** – If direction is not “ASC” or “DESC”.

Examples

```
>>> query.sort_by("price") # Single field, ascending
>>> query.sort_by(("price", "DESC")) # Single field, descending
>>> query.sort_by(["price", "rating"]) # Multiple fields (only first used)
>>> query.sort_by([("price", "DESC"), ("rating", "ASC")])
```

Note

When multiple fields are specified, only the first field is used for sorting in Redis. Future versions may support multi-field sorting through post-query sorting in Python.

timeout (*timeout*)

overrides the timeout parameter of the module

Parameters

timeout (*float*)

Return type

Query

verbatim()

Set the query to be verbatim, i.e., use no query expansion or stemming.

Return type

Query

with_payloads()

Ask the engine to return document payloads.

Return type

Query

with_scores()

Ask the engine to return document search scores.

Return type

Query

property batch_size: int | None

Return the batch size for the query.

Returns

The batch size for the query.

Return type

Optional[int]

property ef_runtime: int | None

Return the EF_RUNTIME parameter for the query.

Returns

The EF_RUNTIME value for the query.

Return type

Optional[int]

property filter: str | *FilterExpression*

The filter expression for the query.

property hybrid_policy: str | None

Return the hybrid policy for the query.

Returns

The hybrid policy for the query.

Return type

Optional[str]

property params: dict[str, Any]

Return the parameters for the query.

Returns

The parameters for the query.

Return type

Dict[str, Any]

property query: BaseQuery

Return self as the query object.

property search_buffer_capacity: int | None

Return the SEARCH_BUFFER_CAPACITY parameter for the query.

Returns

The SEARCH_BUFFER_CAPACITY value for the query.

Return type

Optional[int]

property search_window_size: int | None

Return the SEARCH_WINDOW_SIZE parameter for the query.

Returns

The SEARCH_WINDOW_SIZE value for the query.

Return type

Optional[int]

property use_search_history: str | None

Return the USE_SEARCH_HISTORY parameter for the query.

Returns

The USE_SEARCH_HISTORY value for the query.

Return type

Optional[str]

Note**Runtime Parameters for Performance Tuning**

VectorQuery supports runtime parameters for HNSW and SVS-VAMANA indexes that can be adjusted at query time without rebuilding the index:

HNSW Parameters:

- `ef_runtime`: Controls search accuracy (higher = better recall, slower search)

SVS-VAMANA Parameters:

- `search_window_size`: Size of search window for KNN searches
- `use_search_history`: Whether to use search buffer (OFF/ON/AUTO)
- `search_buffer_capacity`: Tuning parameter for 2-level compression

Example with HNSW runtime parameters:

```
from redisvl.query import VectorQuery

query = VectorQuery(
    vector=[0.1, 0.2, 0.3],
    vector_field_name="embedding",
    num_results=10,
    ef_runtime=150 # Higher for better recall
)
```

Example with SVS-VAMANA runtime parameters:

```
query = VectorQuery(
    vector=[0.1, 0.2, 0.3],
    vector_field_name="embedding",
    num_results=10,
    search_window_size=20,
    use_search_history='ON',
    search_buffer_capacity=30
)
```

VectorRangeQuery

```
class VectorRangeQuery(vector, vector_field_name, return_fields=None, filter_expression=None,
    dtype='float32', distance_threshold=0.2, epsilon=None, search_window_size=None,
    use_search_history=None, search_buffer_capacity=None, num_results=10,
    return_score=True, dialect=2, sort_by=None, in_order=False, hybrid_policy=None,
    batch_size=None, normalize_vector_distance=False)
```

Bases: BaseVectorQuery, BaseQuery

A query for running a filtered vector search based on semantic distance threshold.

Parameters

- **vector** (*List[float]*) – The vector to perform the range query with.
- **vector_field_name** (*str*) – The name of the vector field to search against in the database.
- **return_fields** (*List[str]*) – The declared fields to return with search results.
- **filter_expression** (*Union[str, FilterExpression]*, *optional*) – A filter to apply along with the range query. Defaults to None.
- **dtype** (*str*, *optional*) – The dtype of the vector. Defaults to “float32”.
- **distance_threshold** (*float*) – The threshold for vector distance. A smaller threshold indicates a stricter semantic search. Defaults to 0.2.
- **epsilon** (*Optional[float]*) – The relative factor for vector range queries, setting boundaries for candidates within radius * (1 + epsilon). This controls how extensive the search is beyond the specified radius. Higher values increase recall at the expense of performance. Defaults to None, which uses the index-defined epsilon (typically 0.01).
- **search_window_size** (*Optional[int]*) – The size of the search window for SVS-VAMANA range searches. Increasing this value generally yields more accurate but slower search results. Defaults to None, which uses the index-defined value (typically 10).
- **use_search_history** (*Optional[str]*) – For SVS-VAMANA indexes, controls whether to use the search buffer or entire search history. Options are “OFF”, “ON”, or “AUTO”. “AUTO” is always evaluated internally as “ON”. Using the entire history may yield a slightly better graph at the cost of more search time. Defaults to None, which uses the index-defined value (typically “AUTO”).
- **search_buffer_capacity** (*Optional[int]*) – Tuning parameter for SVS-VAMANA indexes using two-level compression (LVQ<X>x<Y> or LeanVec types). Determines the number of vector candidates to collect in the first level of search before the re-ranking level. Defaults to None, which uses the index-defined value (typically SEARCH_WINDOW_SIZE).

- **num_results** (*int*) – The MAX number of results to return. Defaults to 10.
- **return_score** (*bool*, *optional*) – Whether to return the vector distance. Defaults to True.
- **dialect** (*int*, *optional*) – The Redis Search query dialect. Defaults to 2.
- **sort_by** (*Optional[SortSpec]*) – The field(s) to order the results by. Can be: - str: single field name - Tuple[str, str]: (field_name, “ASC”|“DESC”) - List: list of fields or tuples Note: Only the first field is used for Redis sorting. Defaults to None. Results will be ordered by vector distance.
- **in_order** (*bool*) – Requires the terms in the field to have the same order as the terms in the query filter, regardless of the offsets between them. Defaults to False.
- **hybrid_policy** (*Optional[str]*) – Controls how filters are applied during vector search. Options are “BATCHES” (paginates through small batches of nearest neighbors) or “AD-HOC_BF” (computes scores for all vectors passing the filter). “BATCHES” mode is typically faster for queries with selective filters. “ADHOC_BF” mode is better when filters match a large portion of the dataset. Defaults to None, which lets Redis auto-select the optimal policy.
- **batch_size** (*Optional[int]*) – When hybrid_policy is “BATCHES”, controls the number of vectors to fetch in each batch. Larger values may improve performance at the cost of memory usage. Only applies when hybrid_policy=“BATCHES”. Defaults to None, which lets Redis auto-select an appropriate batch size.
- **normalize_vector_distance** (*bool*) – Redis supports 3 distance metrics: L2 (euclidean), IP (inner product), and COSINE. By default, L2 distance returns an unbounded value. COSINE distance returns a value between 0 and 2. IP returns a value determined by the magnitude of the vector. Setting this flag to true converts COSINE and L2 distance to a similarity score between 0 and 1. Note: setting this flag to true for IP will throw a warning since by definition COSINE similarity is normalized IP.

Raises

TypeError – If filter_expression is not of type redisvl.query.FilterExpression

Note

Learn more about vector range queries: <https://redis.io/docs/latest/develop/ai/search-and-query/vectors/#range-query>

dialect(*dialect*)

Add a dialect field to the query.

- **dialect** - dialect version to execute the query under

Parameters

dialect (*int*)

Return type

Query

expander(*expander*)

Add an expander field to the query.

- **expander** - the name of the expander

Parameters**expander** (*str*)**Return type***Query***in_order()**

Match only documents where the query terms appear in the same order in the document. i.e., for the query “hello world”, we do not match “world hello”

Return type*Query***language**(*language*)

Analyze the query as being in the specified language.

Parameters**language** (*str*) – The language (e.g. *chinese* or *english*)**Return type***Query***limit_fields**(**fields*)

Limit the search to specific TEXT fields only.

- **fields**: Each element should be a string, case sensitive field name

from the defined schema.

Parameters**fields** (*str*)**Return type***Query***limit_ids**(**ids*)

Limit the results to a specific set of pre-known document ids of any length.

Return type*Query***no_content**()

Set the query to only return ids and not the document content.

Return type*Query***no_stopwords**()

Prevent the query from being filtered for stopwords. Only useful in very big queries that you are certain contain no stopwords.

Return type*Query***paging**(*offset, num*)

Set the paging for the query (defaults to 0..10).

- **offset**: Paging offset for the results. Defaults to 0
- **num**: How many results do we want

Parameters

- **offset** (*int*)
- **num** (*int*)

Return type*Query***query_string()**

Return the query string of this query only.

Return type*str***return_fields**(**fields*, *skip_decode=None*)

Set the fields to return with search results.

Parameters

- ***fields** – Variable number of field names to return.
- **skip_decode** (*str* | *list[str]* | *None*) – Optional field name or list of field names that should not be decoded. Useful for binary data like embeddings.

Returns

Returns the query object for method chaining.

Return type*self***Raises**

TypeError – If *skip_decode* is not a string, list, or *None*.

scorer(*scorer*)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Since Redis 8.0 default was changed to *BM25STD*.

Parameters

scorer (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)

Return type*Query***set_batch_size**(*batch_size*)

Set the batch size for the query.

Parameters

batch_size (*int*) – The batch size to use when *hybrid_policy* is “BATCHES”.

Raises

- **TypeError** – If *batch_size* is not an integer
- **ValueError** – If *batch_size* is not positive

set_distance_threshold(*distance_threshold*)

Set the distance threshold for the query.

Parameters

distance_threshold (*float*) – Vector distance threshold.

Raises

- **TypeError** – If *distance_threshold* is not a float or int

- **ValueError** – If `distance_threshold` is negative

set_epsilon(*epsilon*)

Set the epsilon parameter for the range query.

Parameters

epsilon (*float*) – The relative factor for vector range queries, setting boundaries for candidates within $\text{radius} * (1 + \text{epsilon})$.

Raises

- **TypeError** – If `epsilon` is not a float or int
- **ValueError** – If `epsilon` is negative

set_filter(*filter_expression=None*)

Set the filter expression for the query.

Parameters

filter_expression (*Optional[Union[str, FilterExpression]], optional*) – The filter expression or query string to use on the query.

Raises

TypeError – If `filter_expression` is not a valid `FilterExpression` or string.

set_hybrid_policy(*hybrid_policy*)

Set the hybrid policy for the query.

Parameters

hybrid_policy (*str*) – The hybrid policy to use. Options are “BATCHES” or “AD-HOC_BF”.

Raises

ValueError – If `hybrid_policy` is not one of the valid options

set_search_buffer_capacity(*search_buffer_capacity*)

Set the SEARCH_BUFFER_CAPACITY parameter for the range query.

Parameters

search_buffer_capacity (*int*) – Tuning parameter for SVS-VAMANA indexes using two-level compression.

Raises

- **TypeError** – If `search_buffer_capacity` is not an integer
- **ValueError** – If `search_buffer_capacity` is not positive

set_search_window_size(*search_window_size*)

Set the SEARCH_WINDOW_SIZE parameter for the range query.

Parameters

search_window_size (*int*) – The size of the search window for SVS-VAMANA range searches.

Raises

- **TypeError** – If `search_window_size` is not an integer
- **ValueError** – If `search_window_size` is not positive

set_use_search_history(*use_search_history*)

Set the USE_SEARCH_HISTORY parameter for the range query.

Parameters

use_search_history (*str*) – Controls whether to use the search buffer or entire history. Must be one of “OFF”, “ON”, or “AUTO”.

Raises

- **TypeError** – If *use_search_history* is not a string
- **ValueError** – If *use_search_history* is not one of the valid options

slop(*slop*)

Allow a maximum of N intervening non-matched terms between phrase terms (0 means exact phrase).

Parameters

slop (*int*)

Return type

Query

sort_by(*sort_spec=None, asc=True*)

Set the sort order for query results.

This method supports sorting by single or multiple fields. Note that Redis Search natively supports only a single SORTBY field. When multiple fields are specified, only the FIRST field is used for the Redis SORTBY clause.

Parameters

- **sort_spec** (*str | tuple[str, str] | list[str | tuple[str, str]] | None*) – Sort specification in various formats: - *str*: single field name - *Tuple*[*str*, *str*]: (*field_name*, “ASC”|“DESC”) - *List*: list of field names or tuples
- **asc** (*bool*) – Default sort direction when not specified (only used when *sort_spec* is a string). Defaults to True (ascending).

Returns

Returns the query object for method chaining.

Return type

self

Raises

- **TypeError** – If *sort_spec* is not a valid type.
- **ValueError** – If direction is not “ASC” or “DESC”.

Examples

```
>>> query.sort_by("price") # Single field, ascending
>>> query.sort_by(("price", "DESC")) # Single field, descending
>>> query.sort_by(["price", "rating"]) # Multiple fields (only first used)
>>> query.sort_by([("price", "DESC"), ("rating", "ASC")])
```

Note

When multiple fields are specified, only the first field is used for sorting in Redis. Future versions may support multi-field sorting through post-query sorting in Python.

timeout (*timeout*)

overrides the timeout parameter of the module

Parameters

timeout (*float*)

Return type

Query

verbatim()

Set the query to be verbatim, i.e., use no query expansion or stemming.

Return type

Query

with_payloads()

Ask the engine to return document payloads.

Return type

Query

with_scores()

Ask the engine to return document search scores.

Return type

Query

property batch_size: int | None

Return the batch size for the query.

Returns

The batch size for the query.

Return type

Optional[int]

property distance_threshold: float

Return the distance threshold for the query.

Returns

The distance threshold for the query.

Return type

float

property epsilon: float | None

Return the epsilon for the query.

Returns

The epsilon for the query, or None if not set.

Return type

Optional[float]

property filter: `str` | *FilterExpression*

The filter expression for the query.

property hybrid_policy: `str` | `None`

Return the hybrid policy for the query.

Returns

The hybrid policy for the query.

Return type

Optional[str]

property params: `dict[str, Any]`

Return the parameters for the query.

Returns

The parameters for the query.

Return type

Dict[str, Any]

property query: `BaseQuery`

Return self as the query object.

property search_buffer_capacity: `int` | `None`

Return the SEARCH_BUFFER_CAPACITY parameter for the query.

Returns

The SEARCH_BUFFER_CAPACITY value for the query.

Return type

Optional[int]

property search_window_size: `int` | `None`

Return the SEARCH_WINDOW_SIZE parameter for the query.

Returns

The SEARCH_WINDOW_SIZE value for the query.

Return type

Optional[int]

property use_search_history: `str` | `None`

Return the USE_SEARCH_HISTORY parameter for the query.

Returns

The USE_SEARCH_HISTORY value for the query.

Return type

Optional[str]

Note

Runtime Parameters for Range Queries

VectorRangeQuery supports runtime parameters for controlling range search behavior:

HNSW & SVS-VAMANA Parameters:

- `epsilon`: Range search approximation factor (default: 0.01)

SVS-VAMANA Parameters:

- `search_window_size`: Size of search window
- `use_search_history`: Whether to use search buffer (OFF/ON/AUTO)
- `search_buffer_capacity`: Tuning parameter for 2-level compression

Example:

```
from redisvl.query import VectorRangeQuery

query = VectorRangeQuery(
    vector=[0.1, 0.2, 0.3],
    vector_field_name="embedding",
    distance_threshold=0.3,
    epsilon=0.05,           # Approximation factor
    search_window_size=20,   # SVS-VAMANA only
    use_search_history='AUTO' # SVS-VAMANA only
)
```

AggregateHybridQuery

```
class AggregateHybridQuery(text, text_field_name, vector, vector_field_name, text_scorer='BM25STD',
                           filter_expression=None, alpha=0.7, dtype='float32', num_results=10,
                           return_fields=None, stopwords='english', dialect=2, text_weights=None)
```

Bases: `AggregationQuery`

`AggregateHybridQuery` combines text and vector search in Redis. It allows you to perform a hybrid search using both text and vector similarity. It scores documents based on a weighted combination of text and vector similarity.

```
from redisvl.query import AggregateHybridQuery
from redisvl.index import SearchIndex

index = SearchIndex.from_yaml("path/to/index.yaml")

query = AggregateHybridQuery(
    text="example text",
    text_field_name="text_field",
    vector=[0.1, 0.2, 0.3],
    vector_field_name="vector_field",
    text_scorer="BM25STD",
    filter_expression=None,
    alpha=0.7,
    dtype="float32",
    num_results=10,
    return_fields=["field1", "field2"],
    stopwords="english",
    dialect=2,
)

results = index.query(query)
```

Instantiates a `AggregateHybridQuery` object.

Parameters

- **text** (*str*) – The text to search for.
 - **text_field_name** (*str*) – The text field name to search in.
 - **vector** (*Union[bytes, List[float]]*) – The vector to perform vector similarity search.
 - **vector_field_name** (*str*) – The vector field name to search in.
 - **text_scorer** (*str, optional*) – The text scorer to use. Options are {TFIDF, TFIDF.DOCNORM, BM25, DISMAX, DOCSCORE, BM25STD}. Defaults to “BM25STD”.
 - **filter_expression** (*Optional[FilterExpression], optional*) – The filter expression to use. Defaults to None.
 - **alpha** (*float, optional*) – The weight of the vector similarity. Documents will be scored as: $\text{hybrid_score} = (\text{alpha}) * \text{vector_score} + (1 - \text{alpha}) * \text{text_score}$. Defaults to 0.7.
 - **dtype** (*str, optional*) – The data type of the vector. Defaults to “float32”.
 - **num_results** (*int, optional*) – The number of results to return. Defaults to 10.
 - **return_fields** (*Optional[List[str]], optional*) – The fields to return. Defaults to None.
 - **stopwords** (*Optional[Union[str, Set[str]]], optional*) – The stopwords to remove from the provided text prior to search-use. If a string such as “english” “german” is provided then a default set of stopwords for that language will be used. if a list, set, or tuple of strings is provided then those will be used as stopwords. Defaults to “english”. if set to “None” then no stopwords will be removed.
- Note: This parameter controls query-time stopwords filtering (client-side). For index-level stopwords configuration (server-side), see `IndexInfo.stopwords`. Using query-time stopwords with index-level `STOPWORDS 0` is counterproductive.
- **dialect** (*int, optional*) – The Redis dialect version. Defaults to 2.
 - **text_weights** (*Optional[Dict[str, float]]*) – The importance weighting of individual words within the query text. Defaults to None, as no modifications will be made to the `text_scorer` score.

Note

AggregateHybridQuery uses FT.AGGREGATE commands which do NOT support runtime parameters. For runtime parameter support (`ef_runtime`, `search_window_size`, etc.), use `VectorQuery` or `VectorRangeQuery` which use FT.SEARCH commands.

Raises

- **ValueError** – If the text string is empty, or if the text string becomes empty after stopwords are removed.
- **TypeError** – If the stopwords are not a set, list, or tuple of strings.

Parameters

- **text** (*str*)
- **text_field_name** (*str*)
- **vector** (*bytes | list[float]*)
- **vector_field_name** (*str*)

- **text_scorer** (*str*)
- **filter_expression** (*str* | `FilterExpression` | *None*)
- **alpha** (*float*)
- **dtype** (*str*)
- **num_results** (*int*)
- **return_fields** (*list[str]* | *None*)
- **stopwords** (*str* | *set[str]* | *None*)
- **dialect** (*int*)
- **text_weights** (*dict[str, float]* | *None*)

add_scores()

If set, includes the score as an ordinary field of the row.

Return type

AggregateRequest

apply(kwexpr)**

Specify one or more projection expressions to add to each result

Parameters

- **kwexpr:** One or more key-value pairs for a projection. The key is the alias for the projection, and the value is the projection expression itself, for example *apply(square_root=sqrt(@foo))*

Return type

AggregateRequest

dialect(dialect)

Add a dialect field to the aggregate command.

- **dialect** - dialect version to execute the query under

Parameters

dialect (*int*)

Return type

AggregateRequest

filter(expressions)

Specify filter for post-query results using predicates relating to values in the result set.

Parameters

- **fields:** Fields to group by. This can either be a single string, or a list of strings.

Parameters

expressions (*str* | *List[str]*)

Return type

AggregateRequest

group_by(*fields*, **reducers*)

Specify by which fields to group the aggregation.

Parameters

- **fields:** Fields to group by. This can either be a single string, or a list of strings. both cases, the field should be specified as *@field*.
- **reducers:** One or more reducers. Reducers may be found in the *aggregation* module.

Parameters

- **fields** (*str* | *List[str]*)
- **reducers** (*Reducer*)

Return type

AggregateRequest

limit(*offset*, *num*)

Sets the limit for the most recent group or query.

If no group has been defined yet (via *group_by()*) then this sets the limit for the initial pool of results from the query. Otherwise, this limits the number of items operated on from the previous group.

Setting a limit on the initial search results may be useful when attempting to execute an aggregation on a sample of a large data set.

Parameters

- **offset:** Result offset from which to begin paging
- **num:** Number of results to return

Example of sorting the initial results:

```
` AggregateRequest("@sale_amount:[10000, inf]") .limit(0, 10) .
group_by("@state", r.count()) `
```

Will only group by the states found in the first 10 results of the query *@sale_amount:[10000, inf]*. On the other hand,

```
` AggregateRequest("@sale_amount:[10000, inf]") .limit(0, 1000) .
group_by("@state", r.count() .limit(0, 10)) `
```

Will group all the results matching the query, but only return the first 10 groups.

If you only wish to return a *top-N* style query, consider using *sort_by()* instead.

Parameters

- **offset** (*int*)
- **num** (*int*)

Return type

AggregateRequest

load(**fields*)

Indicate the fields to be returned in the response. These fields are returned in addition to any others implicitly specified.

Parameters

- **fields:** If fields not specified, all the fields will be loaded.

Otherwise, fields should be given in the format of *@field*.

Parameters

fields (*str*)

Return type

AggregateRequest

scorer(*scorer*)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Parameters

scorer (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)

Return type

AggregateRequest

set_text_weights(*weights*)

Set or update the text weights for the query.

Parameters

weights (*dict[str, float]*) – Dictionary of word:weight mappings

sort_by(**fields, **kwargs*)

Indicate how the results should be sorted. This can also be used for *top-N* style queries

Parameters

- **fields:** The fields by which to sort. This can be either a single field or a list of fields. If you wish to specify order, you can use the *Asc* or *Desc* wrapper classes.
- **max:** Maximum number of results to return. This can be used instead of *LIMIT* and is also faster.

Example of sorting by *foo* ascending and *bar* descending:

```
` sort_by(Asc("@foo"), Desc("@bar")) `
```

Return the top 10 customers:

```
` AggregateRequest().group_by("@customer", r.sum("@paid").alias(FIELDNAME))  
.sort_by(Desc("@paid"), max=10) `
```

Parameters

fields (*str*)

Return type

AggregateRequest

with_schema()

If set, the *schema* property will contain a list of [*field*, *type*] entries in the result object.

Return type

AggregateRequest

property params: *dict[str, Any]*

Return the parameters for the aggregation.

Returns

The parameters for the aggregation.

Return type

Dict[str, Any]

property stopwords: set[str]

Return the stopwords used in the query. :returns: The stopwords used in the query. :rtype: Set[str]

property text_weights: dict[str, float]

Get the text weights.

Returns

weight mappings.

Return type

Dictionary of word

Note

The `stopwords` parameter in `AggregateHybridQuery` (and `HybridQuery`) controls query-time stopwords filtering (client-side). For index-level stopwords configuration (server-side), see `redisvl.schema.IndexInfo.stopwords`. Using query-time stopwords with index-level `STOPWORDS 0` is counterproductive.

Note

`HybridQuery` and `AggregateHybridQuery` apply linear combination inconsistently. `HybridQuery` uses `linear_alpha` to weight the text score, while `AggregateHybridQuery` uses `alpha` to weight the vector score. When switching between the two classes, take care to revise your `alpha` setting.

Note**Runtime Parameters for Hybrid Queries**

Important: `AggregateHybridQuery` uses `FT.AGGREGATE` commands which do NOT support runtime parameters. Runtime parameters (`ef_runtime`, `search_window_size`, `use_search_history`, `search_buffer_capacity`) are only supported with `FT.SEARCH` commands.

For runtime parameter support, use `HybridQuery`, `VectorQuery`, or `VectorRangeQuery` instead of `AggregateHybridQuery`.

Example with `HybridQuery` (supports runtime parameters):

```
from redisvl.query import HybridQuery

query = HybridQuery(
    text="query string",
    text_field_name="description",
    vector=[0.1, 0.2, 0.3],
    vector_field_name="embedding",
    vector_search_method="KNN",
    knn_ef_runtime=150, # Runtime parameters work with HybridQuery
    return_fields=["description"],
    num_results=10,
)
```

HybridQuery

```
class HybridQuery(text, text_field_name, vector, vector_field_name, vector_param_name='vector',
                  text_scorer='BM25STD', yield_text_score_as=None, vector_search_method=None,
                  knn_ef_runtime=10, range_radius=None, range_epsilon=0.01, yield_vsim_score_as=None,
                  filter_expression=None, combination_method=None, rrf_window=20, rrf_constant=60,
                  linear_alpha=0.3, yield_combined_score_as=None, dtype='float32', num_results=10,
                  return_fields=None, stopwords='english', text_weights=None)
```

Bases: object

A hybrid search query that combines text search and vector similarity, with configurable fusion methods.

```
from redisvl.query import HybridQuery
from redisvl.index import SearchIndex

index = SearchIndex.from_yaml("path/to/index.yaml")

query = HybridQuery(
    text="example text",
    text_field_name="text_field",
    vector=[0.1, 0.2, 0.3],
    vector_field_name="vector_field",
    text_scorer="BM25STD",
    yield_text_score_as="text_score",
    yield_vsim_score_as="vector_similarity",
    combination_method="LINEAR",
    linear_alpha=0.3,
    yield_combined_score_as="hybrid_score",
    num_results=10,
    return_fields=["field1", "field2"],
    stopwords="english",
)

results = index.query(query)
```

➡ See also

- [FT.HYBRID command documentation](#)
- [redis-py hybrid_search documentation](#)

Instantiates a HybridQuery object.

Parameters

- **text** (*str*) – The text to search for.
- **text_field_name** (*str*) – The text field name to search in.
- **vector** (*bytes* | *list[float]*) – The vector to perform vector similarity search.
- **vector_field_name** (*str*) – The vector field name to search in.
- **vector_param_name** (*str*) – The name of the parameter substitution containing the vector blob.

- **text_scorer** (*str*) – The text scorer to use. Options are {TFIDF, TFIDF.DOCNORM, BM25STD, BM25STD.NORM, BM25STD.TANH, DISMAX, DOCSCORE, HAMMING}. Defaults to “BM25STD”. For more information about supported scoring algorithms, see <https://redis.io/docs/latest/develop/ai/search-and-query/advanced-concepts/scoring/>
- **yield_text_score_as** (*str* | *None*) – The name of the field to yield the text score as.
- **vector_search_method** (*Literal*['KNN', 'RANGE'] | *None*) – The vector search method to use. Options are {KNN, RANGE}. Defaults to None.
- **knn_ef_runtime** (*int*) – The exploration factor parameter for HNSW, optional if *vector_search_method* is “KNN”.
- **range_radius** (*float* | *None*) – The search radius to use, required if *vector_search_method* is “RANGE”.
- **range_epsilon** (*float*) – The epsilon value to use, optional if *vector_search_method* is “RANGE”; defines the accuracy of the search.
- **yield_vsim_score_as** (*str* | *None*) – The name of the field to yield the vector similarity score as.
- **filter_expression** (*str* | *FilterExpression* | *None*) – The filter expression to use for both the text and vector searches. Defaults to None.
- **combination_method** (*Literal*['RRF', 'LINEAR'] | *None*) – The combination method to use. Options are {RRF, LINEAR}. If not specified, the server defaults to RRF. If “RRF” is specified, then at least one of *rrf_window* or *rrf_constant* must be provided. If “LINEAR” is specified, then at least one of *linear_alpha* or *linear_beta* must be provided.
- **rrf_window** (*int*) – The window size to use for the reciprocal rank fusion (RRF) combination method. Limits fusion scope.
- **rrf_constant** (*int*) – The constant to use for the reciprocal rank fusion (RRF) combination method. Controls decay of rank influence.
- **linear_alpha** (*float*) – The weight of the text query for the linear combination method (LINEAR).
- **yield_combined_score_as** (*str* | *None*) – The name of the field to yield the combined score as.
- **dtype** (*str*) – The data type of the vector. Defaults to “float32”.
- **num_results** (*int* | *None*) – The number of results to return.
- **return_fields** (*list*[*str*] | *None*) – The fields to return. Defaults to None.
- **stopwords** (*Optional*[*Union*[*str*, *Set*[*str*]]], *optional*) – The stopwords to remove from the provided text prior to search-use. If a string such as “english” “german” is provided then a default set of stopwords for that language will be used. if a list, set, or tuple of strings is provided then those will be used as stopwords. Defaults to “english”. if set to “None” then no stopwords will be removed.

Note: This parameter controls query-time stopwords filtering (client-side). For index-level stopwords configuration (server-side), see `IndexInfo.stopwords`. Using query-time stopwords with index-level STOPWORDS 0 is counterproductive.

- **text_weights** (*Optional*[*Dict*[*str*, *float*]]) – The importance weighting of individual words within the query text. Defaults to None, as no modifications will be made to the *text_scorer* score.

Raises

- **ImportError** – If `redis-py` $\geq 7.1.0$ is not installed.
- **TypeError** – If the stopwords are not a set, list, or tuple of strings.
- **ValueError** – If the text string is empty, or if the text string becomes empty after stopwords are removed.
- **ValueError** – If `vector_search_method` is defined and isn't one of {KNN, RANGE}.
- **ValueError** – If `vector_search_method` is "KNN" and `knn_k` is not provided.
- **ValueError** – If `vector_search_method` is "RANGE" and `range_radius` is not provided.

Note

The stopwords parameter in *HybridQuery* (and *AggregateHybridQuery*) controls query-time stopwords filtering (client-side). For index-level stopwords configuration (server-side), see `redisvl.schema.IndexInfo.stopwords`. Using query-time stopwords with index-level STOPWORDS `0` is counterproductive.

Note

HybridQuery and *AggregateHybridQuery* apply linear combination inconsistently. *HybridQuery* uses `linear_alpha` to weight the text score, while *AggregateHybridQuery* uses `alpha` to weight the vector score. When switching between the two classes, take care to revise your alpha setting.

TextQuery

```
class TextQuery(text, text_field_name, text_scorer='BM25STD', filter_expression=None, return_fields=None,
               num_results=10, return_score=True, dialect=2, sort_by=None, in_order=False, params=None,
               stopwords='english', text_weights=None)
```

Bases: *BaseQuery*

TextQuery is a query for running a full text search, along with an optional filter expression.

```
from redisvl.query import TextQuery
from redisvl.index import SearchIndex

index = SearchIndex.from_yaml("index.yaml")

query = TextQuery(
    text="example text",
    text_field_name="text_field",
    text_scorer="BM25STD",
    filter_expression=None,
    num_results=10,
    return_fields=["field1", "field2"],
    stopwords="english",
    dialect=2,
)

results = index.query(query)
```

A query for running a full text search, along with an optional filter expression.

Parameters

- **text** (*str*) – The text string to perform the text search with.
 - **text_field_name** (*Union[str, Dict[str, float]]*) – The name of the document field to perform text search on, or a dictionary mapping field names to their weights.
 - **text_scorer** (*str, optional*) – The text scoring algorithm to use. Defaults to BM25STD. Options are {TFIDF, BM25STD, BM25, TFIDF.DOCNORM, DIS-MAX, DOCSCORE}. See <https://redis.io/docs/latest/develop/interact/search-and-query/advanced-concepts/scoring/>
 - **filter_expression** (*Union[str, FilterExpression], optional*) – A filter to apply along with the text search. Defaults to None.
 - **return_fields** (*List[str]*) – The declared fields to return with search results.
 - **num_results** (*int, optional*) – The top k results to return from the search. Defaults to 10.
 - **return_score** (*bool, optional*) – Whether to return the text score. Defaults to True.
 - **dialect** (*int, optional*) – The Redis Search query dialect. Defaults to 2.
 - **sort_by** (*Optional[SortSpec]*) – The field(s) to order the results by. Can be: - *str*: single field name - *Tuple[str, str]*: (field_name, “ASC”|”DESC”) - *List*: list of fields or tuples Note: Only the first field is used for Redis sorting. Defaults to None. Results will be ordered by text score.
 - **in_order** (*bool*) – Requires the terms in the field to have the same order as the terms in the query filter, regardless of the offsets between them. Defaults to False.
 - **params** (*Optional[Dict[str, Any]], optional*) – The parameters for the query. Defaults to None.
 - **stopwords** (*Optional[Union[str, Set[str]]]*) – The set of stop words to remove from the query text (client-side filtering). If a language like ‘english’ or ‘spanish’ is provided a default set of stopwords for that language will be used. Users may specify their own stop words by providing a List or Set of words. if set to None, then no words will be removed. Defaults to ‘english’.
- Note: This parameter controls query-time stopwords filtering (client-side). For index-level stopwords configuration (server-side), see `IndexInfo.stopwords`. Using query-time stopwords with index-level STOPWORDS 0 is counterproductive.
- **text_weights** (*Optional[Dict[str, float]]*) – The importance weighting of individual words within the query text. Defaults to None, as no modifications will be made to the text_scorer score.

Raises

- **ValueError** – if stopwords language string cannot be loaded.
- **TypeError** – If stopwords is not a valid iterable set of strings.

dialect (*dialect*)

Add a dialect field to the query.

- **dialect** - dialect version to execute the query under

Parameters

dialect (*int*)

Return type*Query***expander**(*expander*)

Add an expander field to the query.

- **expander** - the name of the expander

Parameters**expander** (*str*)**Return type***Query***in_order**()

Match only documents where the query terms appear in the same order in the document. i.e., for the query “hello world”, we do not match “world hello”

Return type*Query***language**(*language*)

Analyze the query as being in the specified language.

Parameters**language** (*str*) – The language (e.g. *chinese* or *english*)**Return type***Query***limit_fields**(**fields*)

Limit the search to specific TEXT fields only.

- **fields**: Each element should be a string, case sensitive field name from the defined schema.

Parameters**fields** (*str*)**Return type***Query***limit_ids**(**ids*)

Limit the results to a specific set of pre-known document ids of any length.

Return type*Query***no_content**()

Set the query to only return ids and not the document content.

Return type*Query***no_stopwords**()

Prevent the query from being filtered for stopwords. Only useful in very big queries that you are certain contain no stopwords.

Return type*Query*

paging(*offset, num*)

Set the paging for the query (defaults to 0..10).

- **offset**: Paging offset for the results. Defaults to 0
- **num**: How many results do we want

Parameters

- **offset** (*int*)
- **num** (*int*)

Return type

Query

query_string()

Return the query string of this query only.

Return type

str

return_fields(**fields, skip_decode=None*)

Set the fields to return with search results.

Parameters

- ***fields** – Variable number of field names to return.
- **skip_decode** (*str | list[str] | None*) – Optional field name or list of field names that should not be decoded. Useful for binary data like embeddings.

Returns

Returns the query object for method chaining.

Return type

self

Raises

TypeError – If skip_decode is not a string, list, or None.

scorer(*scorer*)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Since Redis 8.0 default was changed to BM25STD.

Parameters

scorer (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)

Return type

Query

set_field_weights(*field_weights*)

Set or update the field weights for the query.

Parameters

field_weights (*str | dict[str, float]*) – Either a single field name or dictionary of field:weight mappings

set_filter(*filter_expression=None*)

Set the filter expression for the query.

Parameters

filter_expression (*Optional[Union[str, FilterExpression]], optional*) – The filter expression or query string to use on the query.

Raises

TypeError – If filter_expression is not a valid FilterExpression or string.

set_text_weights(weights)

Set or update the text weights for the query.

Parameters

- **text_weights** – Dictionary of word:weight mappings
- **weights** (*dict[str, float]*)

slop(slop)

Allow a maximum of N intervening non-matched terms between phrase terms (0 means exact phrase).

Parameters

slop (*int*)

Return type

Query

sort_by(sort_spec=None, asc=True)

Set the sort order for query results.

This method supports sorting by single or multiple fields. Note that Redis Search natively supports only a single SORTBY field. When multiple fields are specified, only the FIRST field is used for the Redis SORTBY clause.

Parameters

- **sort_spec** (*str | tuple[str, str] | list[str | tuple[str, str]] | None*) – Sort specification in various formats: - str: single field name - Tuple[str, str]: (field_name, “ASC”|“DESC”) - List: list of field names or tuples
- **asc** (*bool*) – Default sort direction when not specified (only used when sort_spec is a string). Defaults to True (ascending).

Returns

Returns the query object for method chaining.

Return type

self

Raises

- **TypeError** – If sort_spec is not a valid type.
- **ValueError** – If direction is not “ASC” or “DESC”.

Examples

```
>>> query.sort_by("price") # Single field, ascending
>>> query.sort_by(("price", "DESC")) # Single field, descending
>>> query.sort_by(["price", "rating"]) # Multiple fields (only first used)
>>> query.sort_by([("price", "DESC"), ("rating", "ASC")])
```

Note

When multiple fields are specified, only the first field is used for sorting in Redis. Future versions may support multi-field sorting through post-query sorting in Python.

timeout (*timeout*)

overrides the timeout parameter of the module

Parameters

timeout (*float*)

Return type

Query

verbatim()

Set the query to be verbatim, i.e., use no query expansion or stemming.

Return type

Query

with_payloads()

Ask the engine to return document payloads.

Return type

Query

with_scores()

Ask the engine to return document search scores.

Return type

Query

property field_weights: `dict[str, float]`

Get the field weights for the query.

Returns

Dictionary mapping field names to their weights

property filter: `str | FilterExpression`

The filter expression for the query.

property params: `dict[str, Any]`

Return the query parameters.

property query: `BaseQuery`

Return self as the query object.

property text_field_name: `str | dict[str, float]`

Get the text field name(s) - for backward compatibility.

Returns

Either a single field name string (if only one field with weight 1.0) or a dictionary of field:weight mappings.

property text_weights: dict[str, float]

Get the text weights.

Returns

weight mappings.

Return type

Dictionary of word

Note

The `stopwords` parameter in `TextQuery` controls query-time stopwords filtering (client-side). For index-level stopwords configuration (server-side), see `redisvl.schema.IndexInfo.stopwords`. Using query-time stopwords with index-level STOPWORDS `0` is counterproductive.

FilterQuery

class FilterQuery(*filter_expression=None, return_fields=None, num_results=10, dialect=2, sort_by=None, in_order=False, params=None*)

Bases: BaseQuery

A query for running a filtered search with a filter expression.

Parameters

- **filter_expression** (*Optional[Union[str, FilterExpression]]*) – The optional filter expression to query with. Defaults to `*`.
- **return_fields** (*Optional[List[str]], optional*) – The fields to return.
- **num_results** (*Optional[int], optional*) – The number of results to return. Defaults to 10.
- **dialect** (*int, optional*) – The query dialect. Defaults to 2.
- **sort_by** (*Optional[SortSpec], optional*) – The field(s) to order the results by. Can be: - str: single field name (e.g., “price”) - Tuple[str, str]: (field_name, “ASC”|“DESC”) (e.g., (“price”, “DESC”)) - List: list of fields or tuples (e.g., [“price”, (“rating”, “DESC”)]) Note: Redis Search only supports single-field sorting, so only the first field is used. Defaults to None.
- **in_order** (*bool, optional*) – Requires the terms in the field to have the same order as the terms in the query filter. Defaults to False.
- **params** (*Optional[Dict[str, Any]], optional*) – The parameters for the query. Defaults to None.

Raises

TypeError – If `filter_expression` is not of type `redisvl.query.FilterExpression`

dialect(*dialect*)

Add a dialect field to the query.

- **dialect** - dialect version to execute the query under

Parameters**dialect** (*int*)**Return type***Query***expander**(*expander*)

Add an expander field to the query.

- **expander** - the name of the expander

Parameters**expander** (*str*)**Return type***Query***in_order**()

Match only documents where the query terms appear in the same order in the document. i.e., for the query “hello world”, we do not match “world hello”

Return type*Query***language**(*language*)

Analyze the query as being in the specified language.

Parameters**language** (*str*) – The language (e.g. *chinese* or *english*)**Return type***Query***limit_fields**(**fields*)

Limit the search to specific TEXT fields only.

- **fields**: Each element should be a string, case sensitive field name from the defined schema.

Parameters**fields** (*str*)**Return type***Query***limit_ids**(**ids*)

Limit the results to a specific set of pre-known document ids of any length.

Return type*Query***no_content**()

Set the query to only return ids and not the document content.

Return type*Query*

no_stopwords()

Prevent the query from being filtered for stopwords. Only useful in very big queries that you are certain contain no stopwords.

Return type

Query

paging(*offset*, *num*)

Set the paging for the query (defaults to 0..10).

- **offset**: Paging offset for the results. Defaults to 0
- **num**: How many results do we want

Parameters

- **offset** (*int*)
- **num** (*int*)

Return type

Query

query_string()

Return the query string of this query only.

Return type

str

return_fields(fields*, *skip_decode=None*)**

Set the fields to return with search results.

Parameters

- ***fields** – Variable number of field names to return.
- **skip_decode** (*str* | *list[str]* | *None*) – Optional field name or list of field names that should not be decoded. Useful for binary data like embeddings.

Returns

Returns the query object for method chaining.

Return type

self

Raises

TypeError – If skip_decode is not a string, list, or None.

scorer(*scorer*)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Since Redis 8.0 default was changed to BM25STD.

Parameters

scorer (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)

Return type

Query

set_filter(*filter_expression=None*)

Set the filter expression for the query.

Parameters

filter_expression (*Optional[Union[str, FilterExpression]], optional*) – The filter expression or query string to use on the query.

Raises

TypeError – If filter_expression is not a valid FilterExpression or string.

slop(slop)

Allow a maximum of N intervening non-matched terms between phrase terms (0 means exact phrase).

Parameters

slop (*int*)

Return type

Query

sort_by(sort_spec=None, asc=True)

Set the sort order for query results.

This method supports sorting by single or multiple fields. Note that Redis Search natively supports only a single SORTBY field. When multiple fields are specified, only the FIRST field is used for the Redis SORTBY clause.

Parameters

- **sort_spec** (*str | tuple[str, str] | list[str | tuple[str, str]] | None*) – Sort specification in various formats: - str: single field name - Tuple[str, str]: (field_name, “ASC”|”DESC”) - List: list of field names or tuples
- **asc** (*bool*) – Default sort direction when not specified (only used when sort_spec is a string). Defaults to True (ascending).

Returns

Returns the query object for method chaining.

Return type

self

Raises

- **TypeError** – If sort_spec is not a valid type.
- **ValueError** – If direction is not “ASC” or “DESC”.

Examples

```
>>> query.sort_by("price") # Single field, ascending
>>> query.sort_by(("price", "DESC")) # Single field, descending
>>> query.sort_by(["price", "rating"]) # Multiple fields (only first used)
>>> query.sort_by([("price", "DESC"), ("rating", "ASC")])
```

Note

When multiple fields are specified, only the first field is used for sorting in Redis. Future versions may support multi-field sorting through post-query sorting in Python.

timeout (*timeout*)

overrides the timeout parameter of the module

Parameters

timeout (*float*)

Return type

Query

verbatim()

Set the query to be verbatim, i.e., use no query expansion or stemming.

Return type

Query

with_payloads()

Ask the engine to return document payloads.

Return type

Query

with_scores()

Ask the engine to return document search scores.

Return type

Query

property filter: `str` | *FilterExpression*

The filter expression for the query.

property params: `dict[str, Any]`

Return the query parameters.

property query: `BaseQuery`

Return self as the query object.

CountQuery

class `CountQuery`(*filter_expression=None, dialect=2, params=None*)

Bases: `BaseQuery`

A query for a simple count operation provided some filter expression.

Parameters

- **filter_expression** (*Optional[Union[str, FilterExpression]]*) – The filter expression to query with. Defaults to None.
- **params** (*Optional[Dict[str, Any]], optional*) – The parameters for the query. Defaults to None.
- **dialect** (*int*)

Raises

TypeError – If `filter_expression` is not of type `redisvl.query.FilterExpression`

```
from redisvl.query import CountQuery
from redisvl.query.filter import Tag
```

(continues on next page)

(continued from previous page)

```
t = Tag("brand") == "Nike"
query = CountQuery(filter_expression=t)

count = index.query(query)
```

dialect(*dialect*)

Add a dialect field to the query.

- **dialect** - dialect version to execute the query under

Parameters

dialect (*int*)

Return type

Query

expander(*expander*)

Add an expander field to the query.

- **expander** - the name of the expander

Parameters

expander (*str*)

Return type

Query

in_order()

Match only documents where the query terms appear in the same order in the document. i.e., for the query “hello world”, we do not match “world hello”

Return type

Query

language(*language*)

Analyze the query as being in the specified language.

Parameters

language (*str*) – The language (e.g. *chinese* or *english*)

Return type

Query

limit_fields(**fields*)

Limit the search to specific TEXT fields only.

- **fields**: Each element should be a string, case sensitive field name from the defined schema.

Parameters

fields (*str*)

Return type

Query

limit_ids(*ids)

Limit the results to a specific set of pre-known document ids of any length.

Return type

Query

no_content()

Set the query to only return ids and not the document content.

Return type

Query

no_stopwords()

Prevent the query from being filtered for stopwords. Only useful in very big queries that you are certain contain no stopwords.

Return type

Query

paging(offset, num)

Set the paging for the query (defaults to 0..10).

- **offset**: Paging offset for the results. Defaults to 0
- **num**: How many results do we want

Parameters

- **offset** (*int*)
- **num** (*int*)

Return type

Query

query_string()

Return the query string of this query only.

Return type

str

return_fields(*fields, skip_decode=None)

Set the fields to return with search results.

Parameters

- ***fields** – Variable number of field names to return.
- **skip_decode** (*str* | *list[str]* | *None*) – Optional field name or list of field names that should not be decoded. Useful for binary data like embeddings.

Returns

Returns the query object for method chaining.

Return type

self

Raises

TypeError – If skip_decode is not a string, list, or None.

scorer(*scorer*)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Since Redis 8.0 default was changed to *BM25STD*.

Parameters

scorer (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)

Return type

Query

set_filter(*filter_expression=None*)

Set the filter expression for the query.

Parameters

filter_expression (*Optional[Union[str, FilterExpression]], optional*) – The filter expression or query string to use on the query.

Raises

TypeError – If *filter_expression* is not a valid *FilterExpression* or string.

slop(*slop*)

Allow a maximum of *N* intervening non-matched terms between phrase terms (0 means exact phrase).

Parameters

slop (*int*)

Return type

Query

sort_by(*sort_spec=None, asc=True*)

Set the sort order for query results.

This method supports sorting by single or multiple fields. Note that Redis Search natively supports only a single *SORTBY* field. When multiple fields are specified, only the *FIRST* field is used for the Redis *SORTBY* clause.

Parameters

- **sort_spec** (*str | tuple[str, str] | list[str | tuple[str, str]] | None*) – Sort specification in various formats: - *str*: single field name - *Tuple[str, str]*: (*field_name*, “ASC”|“DESC”) - *List*: list of field names or tuples
- **asc** (*bool*) – Default sort direction when not specified (only used when *sort_spec* is a string). Defaults to *True* (ascending).

Returns

Returns the query object for method chaining.

Return type

self

Raises

- **TypeError** – If *sort_spec* is not a valid type.
- **ValueError** – If direction is not “ASC” or “DESC”.

Examples

```
>>> query.sort_by("price") # Single field, ascending
>>> query.sort_by(("price", "DESC")) # Single field, descending
>>> query.sort_by(["price", "rating"]) # Multiple fields (only first used)
>>> query.sort_by([("price", "DESC"), ("rating", "ASC")])
```

Note

When multiple fields are specified, only the first field is used for sorting in Redis. Future versions may support multi-field sorting through post-query sorting in Python.

timeout (*timeout*)

overrides the timeout parameter of the module

Parameters

timeout (*float*)

Return type

Query

verbatim()

Set the query to be verbatim, i.e., use no query expansion or stemming.

Return type

Query

with_payloads()

Ask the engine to return document payloads.

Return type

Query

with_scores()

Ask the engine to return document search scores.

Return type

Query

property filter: `str` | *FilterExpression*

The filter expression for the query.

property params: `dict[str, Any]`

Return the query parameters.

property query: `BaseQuery`

Return self as the query object.

MultiVectorQuery

class MultiVectorQuery(*vectors*, *return_fields=None*, *filter_expression=None*, *num_results=10*, *dialect=2*)

Bases: `AggregationQuery`

MultiVectorQuery allows for search over multiple vector fields in a document simultaneously. The final score will be a weighted combination of the individual vector similarity scores following the formula:

$$\text{score} = (w_1 * \text{score}_1 + w_2 * \text{score}_2 + w_3 * \text{score}_3 + \dots)$$

Vectors may be of different size and datatype, but must be indexed using the 'cosine' distance_metric.

```
from redisvl.query import MultiVectorQuery, Vector
from redisvl.index import SearchIndex

index = SearchIndex.from_yaml("path/to/index.yaml")

vector_1 = Vector(
    vector=[0.1, 0.2, 0.3],
    field_name="text_vector",
    dtype="float32",
    weight=0.7,
)
vector_2 = Vector(
    vector=[0.5, 0.5],
    field_name="image_vector",
    dtype="bfloat16",
    weight=0.2,
)
vector_3 = Vector(
    vector=[0.1, 0.2, 0.3],
    field_name="text_vector",
    dtype="float64",
    weight=0.5,
)

query = MultiVectorQuery(
    vectors=[vector_1, vector_2, vector_3],
    filter_expression=None,
    num_results=10,
    return_fields=["field1", "field2"],
    dialect=2,
)

results = index.query(query)
```

Instantiates a MultiVectorQuery object.

Parameters

- **vectors** (*Union[Vector, List[Vector]]*) – The Vectors to perform vector similarity search.
- **return_fields** (*Optional[List[str]]*, *optional*) – The fields to return. Defaults to None.

- **filter_expression** (*Optional[Union[str, FilterExpression]]*) – The filter expression to use. Defaults to None.
- **num_results** (*int, optional*) – The number of results to return. Defaults to 10.
- **dialect** (*int, optional*) – The Redis dialect version. Defaults to 2.

add_scores()

If set, includes the score as an ordinary field of the row.

Return type

AggregateRequest

apply(kwexpr)**

Specify one or more projection expressions to add to each result

Parameters

- **kwexpr:** One or more key-value pairs for a projection. The key is the alias for the projection, and the value is the projection expression itself, for example *apply(square_root=sqrt(@foo))*

Return type

AggregateRequest

dialect(dialect)

Add a dialect field to the aggregate command.

- **dialect** - dialect version to execute the query under

Parameters

dialect (*int*)

Return type

AggregateRequest

filter(expressions)

Specify filter for post-query results using predicates relating to values in the result set.

Parameters

- **fields:** Fields to group by. This can either be a single string, or a list of strings.

Parameters

expressions (*str | List[str]*)

Return type

AggregateRequest

group_by(fields, *reducers)

Specify by which fields to group the aggregation.

Parameters

- **fields:** Fields to group by. This can either be a single string, or a list of strings. both cases, the field should be specified as *@field*.
- **reducers:** One or more reducers. Reducers may be found in the *aggregation* module.

Parameters

- **fields** (*str* | *List[str]*)
- **reducers** (*Reducer*)

Return type*AggregateRequest***limit**(*offset*, *num*)

Sets the limit for the most recent group or query.

If no group has been defined yet (via *group_by()*) then this sets the limit for the initial pool of results from the query. Otherwise, this limits the number of items operated on from the previous group.

Setting a limit on the initial search results may be useful when attempting to execute an aggregation on a sample of a large data set.

Parameters

- **offset**: Result offset from which to begin paging
- **num**: Number of results to return

Example of sorting the initial results:

```
` AggregateRequest("@sale_amount:[10000, inf]") .limit(0, 10) .
  group_by("@state", r.count()) `
```

Will only group by the states found in the first 10 results of the query *@sale_amount:[10000, inf]*. On the other hand,

```
` AggregateRequest("@sale_amount:[10000, inf]") .limit(0, 1000) .
  group_by("@state", r.count() .limit(0, 10)) `
```

Will group all the results matching the query, but only return the first 10 groups.

If you only wish to return a *top-N* style query, consider using *sort_by()* instead.

Parameters

- **offset** (*int*)
- **num** (*int*)

Return type*AggregateRequest***load**(**fields*)

Indicate the fields to be returned in the response. These fields are returned in addition to any others implicitly specified.

Parameters

- **fields**: If fields not specified, all the fields will be loaded.

Otherwise, fields should be given in the format of *@field*.

Parameters**fields** (*str*)**Return type***AggregateRequest*

scorer(*scorer*)

Use a different scoring function to evaluate document relevance. Default is *TFIDF*.

Parameters

scorer (*str*) – The scoring function to use (e.g. *TFIDF.DOCNORM* or *BM25*)

Return type

AggregateRequest

sort_by(**fields*, ***kwargs*)

Indicate how the results should be sorted. This can also be used for *top-N* style queries

Parameters

- **fields:** The fields by which to sort. This can be either a single field or a list of fields. If you wish to specify order, you can use the *Asc* or *Desc* wrapper classes.
- **max:** Maximum number of results to return. This can be used instead of *LIMIT* and is also faster.

Example of sorting by *foo* ascending and *bar* descending:

```
` sort_by(Asc("@foo"), Desc("@bar")) `
```

Return the top 10 customers:

```
` AggregateRequest().group_by("@customer", r.sum("@paid").alias(FIELDNAME))  
.sort_by(Desc("@paid"), max=10) `
```

Parameters

fields (*str*)

Return type

AggregateRequest

with_schema()

If set, the *schema* property will contain a list of [*field*, *type*] entries in the result object.

Return type

AggregateRequest

property params: `dict[str, Any]`

Return the parameters for the aggregation.

Returns

The parameters for the aggregation.

Return type

`Dict[str, Any]`

SQLQuery

class SQLQuery(*sql*, *params=None*, *, *sql_redis_options=None*)

Bases: `object`

A query class that translates SQL-like syntax into Redis queries.

This class allows users to write SQL `SELECT` statements that are automatically translated into Redis `FT.SEARCH` or `FT.AGGREGATE` commands.

For TEXT fields with `sql-redis >= 0.4.0`:

- `=` performs exact phrase or exact-term matching

- `LIKE` performs wildcard/pattern matching using SQL `%` wildcards
- `fuzzy(field, 'term')` performs typo-tolerant matching
- `fulltext(field, 'query')` performs tokenized text search

Vector and hybrid search:

- `vector_distance(field, :vec) / cosine_distance(field, :vec)` perform KNN vector similarity search (pass the query vector via `params`)
- `hybrid_vector_search(cosine_distance(field, :vec), fulltext(field, 'query'), rrf())` fuses a text and a vector query server-side via Redis FT.HYBRID (requires `sql-redis >= 0.7.0`, Redis 8.4+, and `redis-py >= 7.1.0`). Use `rrf()` or `linear()` to select the fusion method

```
from redisvl.query import SQLQuery
from redisvl.index import SearchIndex

index = SearchIndex.from_existing("products", redis_url="redis://localhost:6379")

sql_query = SQLQuery('''
    SELECT title, price, category
    FROM products
    WHERE category = 'electronics' AND price < 100
''')

results = index.query(sql_query)
```

Note

Requires the optional `sql-redis` package. Install with: `pip install redisvl[sql-redis]`

Initialize a `SQLQuery`.

Parameters

- **`sql`** (*str*) – The SQL `SELECT` statement to execute.
- **`params`** (*dict[str, Any] | None*) – Optional dictionary of parameters for parameterized queries. Useful for passing vector data for similarity searches.
- **`sql_redis_options`** (*dict[str, Any] | None*) – Optional passthrough options forwarded to `sql-redis` executor creation. Use this to tune how SQL query translation loads and caches index schema metadata. For example, `{"schema_cache_strategy": "lazy"}` loads schemas on demand (the RedisVL default), while `{"schema_cache_strategy": "load_all"}` eagerly loads all schemas up front. These options exist to balance startup cost vs repeated-query performance across many indexes.

Note

`sql-redis >= 0.4.0` uses explicit TEXT search operators. Use `=` for exact phrase matching, `LIKE` for wildcard matching, `fuzzy()` for typo-tolerant matching, and `fulltext()` for tokenized search.

`redis_query_string`(*redis_client=None, redis_url='redis://localhost:6379'*)

Translate the SQL query to a Redis command string.

This method uses the sql-redis translator to convert the SQL statement into the equivalent Redis FT.SEARCH or FT.AGGREGATE command.

Parameters

- **redis_client** (*Any / None*) – A Redis client connection used to load index schemas. If not provided, a connection will be created using redis_url.
- **redis_url** (*str*) – The Redis URL to connect to if redis_client is not provided. Defaults to “redis://localhost:6379”.

Returns

{electronics}”).

Return type

The Redis command string (e.g., ‘FT.SEARCH products “@category

Raises

ImportError – If sql-redis package is not installed.

Example

```
from redisvl.query import SQLQuery

sql_query = SQLQuery("SELECT * FROM products WHERE category = 'electronics'")

# Using redis_url
redis_cmd = sql_query.redis_query_string(redis_url="redis://localhost:6379")

# Or using an existing client
from redis import Redis
client = Redis()
redis_cmd = sql_query.redis_query_string(redis_client=client)

print(redis_cmd)
# Output: FT.SEARCH products "@category:{electronics}"
```

Note

SQLQuery requires the optional sql-redis package. Install with: `pip install redisvl[sql-redis]`

Note

SQLQuery translates SQL SELECT statements into Redis FT.SEARCH or FT.AGGREGATE commands. The SQL syntax supports WHERE clauses, field selection, ordering, and parameterized queries for vector similarity searches.

Note

SQLQuery accepts a sql_redis_options dictionary that is passed through to sql-redis executor creation. The most common option is schema_cache_strategy:

- "lazy" (default) loads schemas on demand, which keeps one-off or narrow queries cheaper.
- "load_all" eagerly loads all schemas up front, which can help when running many SQL queries across many indexes.

Note

SQLQuery supports hybrid search via `hybrid_vector_search(cosine_distance(...), fulltext(...), rrf())`, which translates to a native Redis FT.HYBRID command fusing a text and a vector query server-side. This is the SQL front-end to HybridQuery and requires `sql-redis >= 0.7.0`, Redis 8.4+, and `redis-py >= 7.1.0`.

1.4.5 Filter

FilterExpression

class FilterExpression(*_filter=None, operator=None, left=None, right=None*)

A FilterExpression is a logical combination of filters in RedisVL.

FilterExpressions can be combined using the `&` and `|` operators to create complex expressions that evaluate to the Redis Query language.

This presents an interface by which users can create complex queries without having to know the Redis Query language.

```
from redisvl.query.filter import Tag, Num

brand_is_nike = Tag("brand") == "nike"
price_is_over_100 = Num("price") < 100
f = brand_is_nike & price_is_over_100

print(str(f))

>>> (@brand:{nike} @price:[-inf (100)])
```

This can be combined with the VectorQuery class to create a query:

```
from redisvl.query import VectorQuery

v = VectorQuery(
    vector=[0.1, 0.1, 0.5, ...],
    vector_field_name="product_embedding",
    return_fields=["product_id", "brand", "price"],
    filter_expression=f,
)
```

Note

Filter expressions are typically not called directly. Instead they are built by combining filter statements using the `&` and `|` operators.

Parameters

- **_filter** (*str* / *None*)
- **operator** (*FilterOperator* / *None*)
- **left** (*FilterExpression* / *None*)
- **right** (*FilterExpression* / *None*)

Tag

class Tag(*field*)

A Tag filter can be applied to Tag fields

Parameters

field (*str*)

__eq__ (*other*)

Create a Tag equality filter expression.

Parameters

other (*Union[List[str], str]*) – The tag(s) to filter on.

Return type

FilterExpression

```
from redisvl.query.filter import Tag
```

```
f = Tag("brand") == "nike"
```

__mod__ (*other*)

Create a Tag wildcard filter expression for pattern matching.

This enables wildcard pattern matching on tag fields using the * character. Unlike the equality operator, wildcards are not escaped, allowing patterns with wildcards in any position, such as prefix ("tech*"), suffix ("*tech"), or middle ("*tech*") matches.

Parameters

other (*Union[List[str], str]*) – The tag pattern(s) to filter on. Use * for wildcard matching (e.g., "tech*", "*tech", or "*tech*").

Return type

FilterExpression

```
from redisvl.query.filter import Tag
```

```
f = Tag("category") % "tech*"           # Prefix match
f = Tag("category") % "*tech"           # Suffix match
f = Tag("category") % "*tech*"          # Contains match
f = Tag("category") % "elec*|*soft"      # Multiple wildcard patterns
f = Tag("category") % ["tech*", "*science"] # List of patterns
```

__ne__ (*other*)

Create a Tag inequality filter expression.

Parameters

other (*Union[List[str], str]*) – The tag(s) to filter on.

Return type

FilterExpression

```
from redisvl.query.filter import Tag
f = Tag("brand") != "nike"
```

__str__()

Return the Redis Query string for the Tag filter

Return type

str

Text**class Text(*field*)**

A Text is a FilterField representing a text field in a Redis index.

Note

== and != match the value as a quoted phrase. Any " in the value becomes a space first, so the value cannot close that phrase; a quote already separates tokens at index time, so this matches the same documents that escaping it never could. A value of nothing but quotes therefore becomes an empty phrase, which == matches no document against. % is the pattern operator and interpolates its value untouched.

Parameters**field** (*str*)**__eq__(*other*)**

Create a Text equality filter expression. These expressions yield filters that enforce an exact match on the supplied term(s).

Parameters**other** (*str*) – The text value to filter on.**Return type**

FilterExpression

```
from redisvl.query.filter import Text
f = Text("job") == "engineer"
```

__mod__(*other*)

Create a Text “LIKE” filter expression. A flexible expression that yields filters that can use a variety of additional operators like wildcards (*), fuzzy matches (%%), or combinatorics (!) of the supplied term(s).

Parameters**other** (*str*) – The text value to filter on.**Return type**

FilterExpression

```
from redisvl.query.filter import Text
f = Text("job") % "engine*"           # suffix wild card match
```

(continues on next page)

(continued from previous page)

```
f = Text("job") % "%%engine%%"      # fuzzy match w/ Levenshtein Distance
f = Text("job") % "engineer|doctor"  # contains either term in field
f = Text("job") % "engineer doctor"  # contains both terms in field
```

Note

The value is interpolated raw, which is what makes *, %% and | work. A value carrying a) therefore closes this clause and has its remainder parsed as query syntax, past any surrounding filter. Pass only patterns your own code composes; for a value you did not construct, use ==, which matches it as a literal phrase.

__ne__(other)

Create a Text inequality filter expression. These expressions yield negated filters on exact matches on the supplied term(s). Opposite of an equality filter expression.

Parameters

other (*str*) – The text value to filter on.

Return type

FilterExpression

```
from redisvl.query.filter import Text

f = Text("job") != "engineer"
```

__str__()

Return the Redis Query string for the Text filter

Return type

str

Num**class Num(field)**

A Num is a FilterField representing a numeric field in a Redis index.

Parameters

field (*str*)

__eq__(other)

Create a Numeric equality filter expression.

Parameters

other (*Union[int, float]*) – The value to filter on.

Return type

FilterExpression

```
from redisvl.query.filter import Num

f = Num("zipcode") == 90210
```

__ge__(other)

Create a Numeric greater than or equal to filter expression.

Parameters**other** (*Union[int, float]*) – The value to filter on.**Return type***FilterExpression*

```
from redisvl.query.filter import Num
f = Num("age") >= 18
```

__gt__(other)

Create a Numeric greater than filter expression.

Parameters**other** (*Union[int, float]*) – The value to filter on.**Return type***FilterExpression*

```
from redisvl.query.filter import Num
f = Num("age") > 18
```

__le__(other)

Create a Numeric less than or equal to filter expression.

Parameters**other** (*Union[int, float]*) – The value to filter on.**Return type***FilterExpression*

```
from redisvl.query.filter import Num
f = Num("age") <= 18
```

__lt__(other)

Create a Numeric less than filter expression.

Parameters**other** (*Union[int, float]*) – The value to filter on.**Return type***FilterExpression*

```
from redisvl.query.filter import Num
f = Num("age") < 18
```

__ne__(other)

Create a Numeric inequality filter expression.

Parameters**other** (*Union[int, float]*) – The value to filter on.**Return type***FilterExpression*

```
from redisvl.query.filter import Num

f = Num("zipcode") != 90210
```

`__str__()`

Return the Redis Query string for the Numeric filter

Return type

str

between(*start*, *end*, *inclusive*='both')

Operator for searching values between two numeric values.

Parameters

- **start** (*Union[int, float]*) – The lower bound of the range.
- **end** (*Union[int, float]*) – The upper bound of the range.
- **inclusive** (*str*, *optional*) – Which bounds to include: “both”, “neither”, “left” or “right”. Defaults to “both”.

Raises

- **TypeError** – If either bound is not an int, a float, or another numbers.Real. numpy scalars qualify; Decimal and str do not.
- **ValueError** – If either bound is NaN, or if *inclusive* is not one of the four accepted values.

Return type

FilterExpression

```
from redisvl.query.filter import Num

f = Num("age").between(18, 65)
f = Num("age").between(18, 65, inclusive="neither")
```

Geo

class Geo(*field*)

A Geo is a FilterField representing a geographic (longitude, latitude) field in a Redis index.

Note

Redis indexes latitudes only within +/-85.05112878 degrees (EPSG:900913). A document or a query center nearer a pole than that is silently excluded: the query returns no error and no results, at any radius.

Parameters

field (*str*)

`__eq__`(*other*)

Create a geographic filter within a specified GeoRadius.

Parameters

other (*GeoRadius*) – The geographic spec to filter on.

Return type

FilterExpression

```
from redisvl.query.filter import Geo, GeoRadius

f = Geo("location") == GeoRadius(-122.4194, 37.7749, 1, unit="m")
```

__ne__(other)

Create a geographic filter outside of a specified GeoRadius.

Parameters

other (GeoRadius) – The geographic spec to filter on.

Return type

FilterExpression

```
from redisvl.query.filter import Geo, GeoRadius

f = Geo("location") != GeoRadius(-122.4194, 37.7749, 1, unit="m")
```

__str__()

Return the Redis Query string for the Geo filter

Return type

str

GeoRadius

class GeoRadius(*longitude, latitude, radius=1, unit='km'*)

A GeoRadius is a GeoSpec representing a geographic radius.

Create a GeoRadius specification (GeoSpec)

Parameters

- **longitude** (*float*) – The longitude of the center of the radius, in degrees, from -180 to 180.
- **latitude** (*float*) – The latitude of the center of the radius, in degrees, from -90 to 90.
- **radius** (*float, optional*) – The radius of the circle, in **unit**, greater than 0. Fractional radii are sent as given, so 0.5 with a unit of “km” is half a kilometre. Defaults to 1.
- **unit** (*str, optional*) – The unit of the radius. Defaults to “km”.

Raises

- **TypeError** – If a coordinate or the radius is not an **int**, a **float**, or another numbers. Real. numpy scalars qualify; **Decimal** and **str** do not.
- **ValueError** – If a coordinate is NaN, infinite, or outside its range, if the radius is NaN, infinite, or not greater than 0, or if the unit is not a string spelling one of “m”, “km”, “mi”, or “ft”.

__init__(*longitude, latitude, radius=1, unit='km'*)

Create a GeoRadius specification (GeoSpec)

Parameters

- **longitude** (*float*) – The longitude of the center of the radius, in degrees, from -180 to 180.
- **latitude** (*float*) – The latitude of the center of the radius, in degrees, from -90 to 90.
- **radius** (*float*, *optional*) – The radius of the circle, in **unit**, greater than 0. Fractional radii are sent as given, so 0.5 with a unit of “km” is half a kilometre. Defaults to 1.
- **unit** (*str*, *optional*) – The unit of the radius. Defaults to “km”.

Raises

- **TypeError** – If a coordinate or the radius is not an **int**, a **float**, or another **numbers**. **Real**. **numpy** scalars qualify; **Decimal** and **str** do not.
- **ValueError** – If a coordinate is **NaN**, infinite, or outside its range, if the radius is **NaN**, infinite, or not greater than 0, or if the unit is not a string spelling one of “m”, “km”, “mi”, or “ft”.

Timestamp

class **Timestamp**(*field*)

A timestamp filter for querying date/time fields in Redis.

This filter can handle various date and time formats, including: - datetime objects (with or without timezone) - date objects - ISO-8601 formatted strings - Unix timestamps (as integers or floats)

All timestamps are converted to Unix timestamps in UTC for consistency. Bare date values and date-only ISO strings are anchored to the UTC calendar day, not the host’s local day, and naive datetimes are read as UTC.

Parameters

field (*str*)

__eq__(*other*)

Filter for timestamps equal to the specified value. For date objects (without time), this matches the entire UTC calendar day, from 00:00:00 to 23:59:59.999999 UTC.

Parameters

other (*datetime* | *date* | *str* | *int* | *float*) – A datetime, date, ISO string, or Unix timestamp

Returns

The filter object for method chaining

Return type

self

__ge__(*other*)

Filter for timestamps greater than or equal to the specified value.

For a bare date (or date-only ISO string), this means from the *start* of that UTC day, so the day itself is included.

Parameters

other – A datetime, date, ISO string, or Unix timestamp

Returns

The filter object for method chaining

Return type

self

__gt__(*other*)

Filter for timestamps greater than the specified value.

For a bare date (or date-only ISO string), this means after the *end* of that UTC day, so the day itself is excluded.

Parameters

other – A datetime, date, ISO string, or Unix timestamp

Returns

The filter object for method chaining

Return type

self

__le__(*other*)

Filter for timestamps less than or equal to the specified value.

For a bare date (or date-only ISO string), this means through the *end* of that UTC day, so the day itself is included.

Parameters

other – A datetime, date, ISO string, or Unix timestamp

Returns

The filter object for method chaining

Return type

self

__lt__(*other*)

Filter for timestamps less than the specified value.

For a bare date (or date-only ISO string), this means before the *start* of that UTC day, so the day itself is excluded.

Parameters

other – A datetime, date, ISO string, or Unix timestamp

Returns

The filter object for method chaining

Return type

self

__ne__(*other*)

Filter for timestamps not equal to the specified value. For date objects (without time), this excludes the entire UTC calendar day, from 00:00:00 to 23:59:59.999999 UTC.

Parameters

other (*datetime* | *date* | *str* | *int* | *float*) – A datetime, date, ISO string, or Unix timestamp

Returns

The filter object for method chaining

Return type

self

between(*start*, *end*, *inclusive*='both')

Filter for timestamps between start and end (inclusive).

Bare dates (and date-only ISO strings) span whole UTC calendar days: start anchors to 00:00:00 of its day and end to 23:59:59.999999 of its day, so both endpoint days are covered in full.

Parameters

- **start** – A datetime, date, ISO string, or Unix timestamp
- **end** – A datetime, date, ISO string, or Unix timestamp
- **inclusive** (*str*) – Which endpoints to include – “both” (default), “left”, “right”, or “neither”.

Returns

The filter object for method chaining

Return type

self

1.4.6 Vectorizers

Note

Backwards Compatibility: Several vectorizers have deprecated aliases available in the `redisvl.utils.vectorize.text` module for backwards compatibility:

- `VoyageAITextVectorizer` → Use `VoyageAIVectorizer` instead
- `VertexAITextVectorizer` → Use `GoogleGenAIVectorizer` instead (`VertexAIVectorizer` is itself deprecated; see below)
- `BedrockTextVectorizer` → Use `BedrockVectorizer` instead
- `CustomTextVectorizer` → Use `CustomVectorizer` instead

These aliases are deprecated as of version 0.13.0 and will be removed in a future major release.

HFTextVectorizer

```
class HFTextVectorizer(model='sentence-transformers/all-mpnet-base-v2', dtype='float32', cache=None, *,
                      dims=None)
```

Bases: `BaseVectorizer`

The `HFTextVectorizer` class leverages Hugging Face’s Sentence Transformers for generating vector embeddings from text input.

This vectorizer is particularly useful in scenarios where advanced natural language processing and understanding are required, and ideal for running on your own hardware without usage fees.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

Utilizing this vectorizer involves specifying a pre-trained model from Hugging Face’s vast collection of Sentence Transformers. These models are trained on a variety of datasets and tasks, ensuring versatility and robust performance across different embedding needs.

Note

Some multimodal models can make use of sentence-transformers by passing PIL Image objects in place of strings (e.g. CLIP). To enable those use cases, this class follows the SentenceTransformer convention of hinting that it expects string inputs, but never enforcing it.

Requirements:

- The *sentence-transformers* library must be installed with pip.

```
# Basic usage
vectorizer = HFTextVectorizer(model="sentence-transformers/all-mpnet-base-v2")
embedding = vectorizer.embed("Hello, world!")

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="my_embeddings_cache")

vectorizer = HFTextVectorizer(
    model="sentence-transformers/all-mpnet-base-v2",
    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed("Hello, world!")

# Second call will retrieve from cache
embedding2 = vectorizer.embed("Hello, world!")

# Batch processing
embeddings = vectorizer.embed_many(
    ["Hello, world!", "How are you?"],
    batch_size=2
)

# Multimodal usage
from PIL import Image
vectorizer = HFTextVectorizer(model="sentence-transformers/clip-ViT-L-14")
embeddings1 = vectorizer.embed("Hello, world!")
embeddings2 = vectorizer.embed(Image.open("path/to/your/image.jpg"))
```

Initialize the Hugging Face text vectorizer.

Parameters

- **model** (*str*) – The pre-trained model from Hugging Face’s Sentence Transformers to be used for embedding. Defaults to ‘sentence-transformers/all-mpnet-base-v2’.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional *EmbeddingsCache* instance to cache embeddings for better performance with repeated texts. Defaults to *None*.
- ****kwargs** – Additional parameters to pass to the SentenceTransformer constructor.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the sentence-transformers library is not installed.
- **ValueError** – If there is an error setting the embedding model dimensions.
- **ValueError** – If an invalid dtype is provided.

model_post_init(*context*, /)

This function is meant to behave like a BaseModel method to initialize private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** (*BaseModel*) – The BaseModel instance.
- **context** (*Any*) – The context.

Return type

None

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

property type: `str`

Return the type of vectorizer.

OpenAITextVectorizer

class OpenAITextVectorizer(*model='text-embedding-ada-002', api_config=None, dtype='float32', cache=None, *, dims=None*)

Bases: `BaseVectorizer`

The OpenAITextVectorizer class utilizes OpenAI's API to generate embeddings for text data.

This vectorizer is designed to interact with OpenAI's embeddings API, requiring an API key for authentication. The key can be provided directly in the *api_config* dictionary or through the *OPENAI_API_KEY* environment variable. Users must obtain an API key from OpenAI's website (<https://api.openai.com/>). Additionally, the *openai* python client must be installed with *pip install openai>=1.13.0*.

The vectorizer supports both synchronous and asynchronous operations, allowing for batch processing of texts and flexibility in handling preprocessing tasks.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

```
# Basic usage with OpenAI embeddings
vectorizer = OpenAITextVectorizer(
    model="text-embedding-ada-002",
    api_config={"api_key": "your_api_key"} # OR set OPENAI_API_KEY in your env
)
embedding = vectorizer.embed("Hello, world!")

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="openai_embeddings_cache")

vectorizer = OpenAITextVectorizer(
```

(continues on next page)

(continued from previous page)

```

model="text-embedding-ada-002",
api_config={"api_key": "your_api_key"},
cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed("Hello, world!")

# Second call will retrieve from cache
embedding2 = vectorizer.embed("Hello, world!")

# Asynchronous batch embedding of multiple texts
embeddings = await vectorizer.aembed_many(
    ["Hello, world!", "How are you?"],
    batch_size=2
)

```

Initialize the OpenAI vectorizer.

Parameters

- **model** (*str*) – Model to use for embedding. Defaults to ‘text-embedding-ada-002’.
- **api_config** (*Optional[Dict]*, *optional*) – Dictionary containing the API key and any additional OpenAI API options. Defaults to None.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional *EmbeddingsCache* instance to cache embeddings for better performance with repeated texts. Defaults to None.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the *openai* library is not installed.
- **ValueError** – If the OpenAI API key is not provided.
- **ValueError** – If an invalid dtype is provided.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

property type: `str`

Return the type of vectorizer.

AzureOpenAITextVectorizer

```
class AzureOpenAITextVectorizer(model='text-embedding-ada-002', api_config=None, dtype='float32',
                                cache=None, *, dims=None)
```

Bases: BaseVectorizer

The AzureOpenAITextVectorizer class utilizes AzureOpenAI's API to generate embeddings for text data.

This vectorizer is designed to interact with AzureOpenAI's embeddings API, requiring an API key, an Azure-OpenAI deployment endpoint and API version. These values can be provided directly in the *api_config* dictionary with the parameters 'azure_endpoint', 'api_version' and 'api_key' or through the environment variables 'AZURE_OPENAI_ENDPOINT', 'OPENAI_API_VERSION', and 'AZURE_OPENAI_API_KEY'. Users must obtain these values from the 'Keys and Endpoints' section in their Azure OpenAI service. Additionally, the *openai* python client must be installed with *pip install openai>=1.13.0*.

The vectorizer supports both synchronous and asynchronous operations, allowing for batch processing of texts and flexibility in handling preprocessing tasks.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

```
# Basic usage
vectorizer = AzureOpenAITextVectorizer(
    model="text-embedding-ada-002",
    api_config={
        "api_key": "your_api_key", # OR set AZURE_OPENAI_API_KEY in your env
        "api_version": "your_api_version", # OR set OPENAI_API_VERSION in your env
        "azure_endpoint": "your_azure_endpoint", # OR set AZURE_OPENAI_ENDPOINT in
↪ your env
    }
)
embedding = vectorizer.embed("Hello, world!")

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="azureopenai_embeddings_cache")

vectorizer = AzureOpenAITextVectorizer(
    model="text-embedding-ada-002",
    api_config={
        "api_key": "your_api_key",
        "api_version": "your_api_version",
        "azure_endpoint": "your_azure_endpoint",
    },
    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed("Hello, world!")

# Second call will retrieve from cache
embedding2 = vectorizer.embed("Hello, world!")

# Asynchronous batch embedding of multiple texts
embeddings = await vectorizer.aembed_many(
    ["Hello, world!", "How are you?"],
```

(continues on next page)

(continued from previous page)

```
batch_size=2
)
```

Initialize the AzureOpenAI vectorizer.

Parameters

- **model** (*str*) – Deployment to use for embedding. Must be the ‘Deployment name’ not the ‘Model name’. Defaults to ‘text-embedding-ada-002’.
- **api_config** (*Optional[Dict]*, *optional*) – Dictionary containing the API key, API version, Azure endpoint, and any other API options. Defaults to None.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional EmbeddingsCache instance to cache embeddings for better performance with repeated texts. Defaults to None.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the openai library is not installed.
- **ValueError** – If the AzureOpenAI API key, version, or endpoint are not provided.
- **ValueError** – If an invalid dtype is provided.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

property type: `str`

Return the type of vectorizer.

VertexAIVectorizer

Note

VertexAIVectorizer is **deprecated**. It uses Google’s Vertex AI model-garden SDK, which Google has deprecated with a scheduled removal. Use [GoogleGenAIVectorizer](#) for text embeddings on the supported google-genai SDK. The alias `VertexAITextVectorizer` (in `redisvl.utils.vectorize.text`) is likewise deprecated. Multimodal (image/video) migration is tracked in [issue #620](#).

class VertexAIVectorizer(*model='textembedding-gecko', api_config=None, dtype='float32', cache=None, *, dims=None*)

Bases: `BaseVectorizer`

The VertexAIVectorizer uses Google’s VertexAI embedding model API to create embeddings.

This vectorizer is tailored for use in environments where integration with Google Cloud Platform (GCP) services is a key requirement.

Utilizing this vectorizer requires an active GCP project and location (region), along with appropriate application credentials. These can be provided through the *api_config* dictionary or set the

GOOGLE_APPLICATION_CREDENTIALS env var. Additionally, the vertexai python client must be installed with `pip install google-cloud-aiplatform>=1.26`.

You can optionally enable caching to improve performance when generating embeddings for repeated inputs.

```
# Basic usage
vectorizer = VertexAIVectorizer(
    model="textembedding-gecko",
    api_config={
        "project_id": "your_gcp_project_id", # OR set GCP_PROJECT_ID
        "location": "your_gcp_location",     # OR set GCP_LOCATION
    })
embedding = vectorizer.embed("Hello, world!")

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="vertexai_embeddings_cache")

vectorizer = VertexAIVectorizer(
    model="textembedding-gecko",
    api_config={
        "project_id": "your_gcp_project_id",
        "location": "your_gcp_location",
    },
    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed("Hello, world!")

# Second call will retrieve from cache
embedding2 = vectorizer.embed("Hello, world!")

# Batch embedding of multiple texts
embeddings = vectorizer.embed_many(
    ["Hello, world!", "Goodbye, world!"],
    batch_size=2
)

# Multimodal usage
from vertexai.vision_models import Image, Video

vectorizer = VertexAIVectorizer(
    model="multimodalembding@001",
    api_config={
        "project_id": "your_gcp_project_id", # OR set GCP_PROJECT_ID
        "location": "your_gcp_location",     # OR set GCP_LOCATION
    }
)
text_embedding = vectorizer.embed("Hello, world!")
image_embedding = vectorizer.embed(Image.load_from_file("path/to/your/image.jpg"))
video_embedding = vectorizer.embed(Video.load_from_file("path/to/your/video.mp4"))
```

Initialize the VertexAI vectorizer.

Parameters

- **model** (*str*) – Model to use for embedding. Defaults to ‘textembedding-gecko’.
- **api_config** (*Optional[Dict]*, *optional*) – Dictionary containing the API config details. Defaults to None.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional EmbeddingsCache instance to cache embeddings for better performance with repeated texts. Defaults to None.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the google-cloud-aiplatform library is not installed.
- **ValueError** – If the API key is not provided.
- **ValueError** – If an invalid dtype is provided.

embed_image(*image_path*, ***kwargs*)

Embed an image (from its path on disk) using a VertexAI multimodal model.

Parameters

image_path (*str*)

Return type

list[float] | bytes

embed_video(*video_path*, ***kwargs*)

Embed a video (from its path on disk) using a VertexAI multimodal model.

Parameters

video_path (*str*)

Return type

list[float] | bytes

property is_multimodal: bool

Whether a multimodal model has been configured.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

property type: str

Return the type of vectorizer.

GoogleGenAIVectorizer

```
class GoogleGenAIVectorizer(model='gemini-embedding-001', api_config=None, dtype='float32',
                           cache=None, task_type=None, output_dimensionality=None, *, dims=None)
```

Bases: BaseVectorizer

The GoogleGenAIVectorizer creates embeddings with Google's *google-genai* SDK, the supported replacement for the deprecated Vertex AI model-garden SDK used by *VertexAIVectorizer*.

One client, two backends. *google-genai* reaches both of Google's embedding backends; this vectorizer auto-selects one from your config/environment (or an explicit override):

- **Vertex AI / Gemini Enterprise** (GCP project auth) — for existing Vertex users.
- **Gemini Developer API** (a single API key) — the simplest way to get started.

Credentials are resolved in this order (explicit beats ambient):

1. Explicit override — `api_config={"backend": "vertex" | "gemini"}`.
2. Explicit creds in `api_config` — `api_key` selects Gemini; `project_id` (+ `location`) selects Vertex.
3. Environment — a project (`GOOGLE_CLOUD_PROJECT` | `GCP_PROJECT_ID` with `GOOGLE_CLOUD_LOCATION` | `GCP_LOCATION` and Application Default Credentials via `GOOGLE_APPLICATION_CREDENTIALS`) selects Vertex; otherwise `GEMINI_API_KEY` | `GOOGLE_API_KEY` selects Gemini. If both a project and a key are present, Vertex wins.

Install the client with `pip install redisvl[google-genai]`.

Note

The default model `gemini-embedding-001` returns **3072-dimensional** vectors (4× the width of the legacy `textembedding-gecko`, so 4× the index memory). A reduced `output_dimensionality` returns shorter vectors that Google does **not** re-normalize; this is invisible under the COSINE metric (scale-invariant) but matters for inner-product / L2 — normalize yourself if your index needs it. Embeddings from different models (or dimensions) are not interchangeable — reindex when you change them.

```
# Vertex AI backend
from redisvl.utils.vectorize import GoogleGenAIVectorizer

vectorizer = GoogleGenAIVectorizer(
    model="gemini-embedding-001",
    api_config={
        "project_id": "your-gcp-project", # or set GOOGLE_CLOUD_PROJECT
        "location": "us-central1",       # or set GOOGLE_CLOUD_LOCATION
    },
)
embedding = vectorizer.embed("Hello, world!")

# Gemini Developer API backend
vectorizer = GoogleGenAIVectorizer(
    model="gemini-embedding-001",
    api_config={"api_key": "your-gemini-api-key"}, # or set GEMINI_API_KEY
)

# Reduced dimensions (Matryoshka) + caching
```

(continues on next page)

(continued from previous page)

```

from redisvl.extensions.cache.embeddings import EmbeddingsCache

vectorizer = GoogleGenAIVectorizer(
    model="gemini-embedding-001",
    output_dimensionality=768,
    cache=EmbeddingsCache(name="genai_embeddings_cache"),
)

# Asynchronous batch embedding
embeddings = await vectorizer.aembed_many(
    ["Hello, world!", "How are you?"], batch_size=2
)

```

Initialize the Google GenAI vectorizer.

Parameters

- **model** (*str*) – The embedding model to use. Defaults to ‘gemini-embedding-001’, which works on both backends.
- **api_config** (*Optional[Dict]*) – Auth/client configuration. Recognized keys: `backend` (“vertex”|“gemini”), `project_id`, `location`, `credentials` (Vertex), and `api_key` (Gemini). Model-behavior options are the `task_type`/`output_dimensionality` arguments below, not `api_config` keys. Defaults to None.
- **dtype** (*str*) – The default datatype to use when embedding text as byte arrays. Used when setting `as_buffer=True`. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional cache for repeated inputs.
- **task_type** (*Optional[str]*) – Default embedding task type (e.g. ‘RETRIEVAL_DOCUMENT’, ‘RETRIEVAL_QUERY’). Overridable per call.
- **output_dimensionality** (*Optional[int]*) – Request shorter (Matryoshka) embeddings of this width; sets dims accordingly. Fixed for the vectorizer’s lifetime (cannot be overridden per call). Note Google does not re-normalize reduced vectors.
- ****kwargs** – Additional arguments forwarded to `google.genai.Client` (e.g. `http_options`).
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the google-genai library is not installed.
- **ValueError** – If a backend cannot be resolved, or an invalid dtype is given.

property backend: *str*

‘vertex’ or ‘gemini’.

Type

The resolved backend

model_config: *ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}*

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

property type: `str`

Return the type of vectorizer.

CohereTextVectorizer

class CohereTextVectorizer(*model='embed-english-v3.0', api_config=None, dtype='float32', cache=None, *, dims=None*)

Bases: `BaseVectorizer`

The CohereTextVectorizer class utilizes Cohere's API to generate embeddings for text data.

This vectorizer is designed to interact with Cohere's /embed API, requiring an API key for authentication. The key can be provided directly in the *api_config* dictionary or through the *COHERE_API_KEY* environment variable. User must obtain an API key from Cohere's website (<https://dashboard.cohere.com/>). Additionally, the *cohere* python client must be installed with *pip install cohere*.

The vectorizer supports only synchronous operations, allows for batch processing of texts and flexibility in handling preprocessing tasks.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

```
from redisvl.utils.vectorize import CohereTextVectorizer

# Basic usage
vectorizer = CohereTextVectorizer(
    model="embed-english-v3.0",
    api_config={"api_key": "your-cohere-api-key"} # OR set COHERE_API_KEY in your
    ↪ env
)
query_embedding = vectorizer.embed(
    text="your input query text here",
    input_type="search_query"
)
doc_embeddings = vectorizer.embed_many(
    texts=["your document text", "more document text"],
    input_type="search_document"
)

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="cohere_embeddings_cache")

vectorizer = CohereTextVectorizer(
    model="embed-english-v3.0",
    api_config={"api_key": "your-cohere-api-key"},
    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed(
    text="your input query text here",
    input_type="search_query"
)
```

(continues on next page)

(continued from previous page)

```
# Second call will retrieve from cache
embedding2 = vectorizer.embed(
    text="your input query text here",
    input_type="search_query"
)
```

Initialize the Cohere vectorizer.

Visit <https://cohere.ai/embed> to learn about embeddings.

Parameters

- **model** (*str*) – Model to use for embedding. Defaults to ‘embed-english-v3.0’.
- **api_config** (*Optional[Dict]*, *optional*) – Dictionary containing the API key. Defaults to None.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. ‘float32’ will use Cohere’s float embeddings, ‘int8’ and ‘uint8’ will map to Cohere’s corresponding embedding types. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional EmbeddingsCache instance to cache embeddings for better performance with repeated texts. Defaults to None.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the cohere library is not installed.
- **ValueError** – If the API key is not provided.
- **ValueError** – If an invalid dtype is provided.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

property type: `str`

Return the type of vectorizer.

BedrockVectorizer

Note

For backwards compatibility, an alias `BedrockTextVectorizer` is available in the `redisvl.utils.vectorize.text` module. This alias is deprecated as of version 0.13.0 and will be removed in a future major release.

```
class BedrockVectorizer(model='amazon.titan-embed-text-v2:0', api_config=None, dtype='float32',
                        cache=None, *, dims=None)
```

Bases: `BaseVectorizer`

The `BedrockVectorizer` class utilizes Amazon Bedrock’s API to generate embeddings for text or image data.

This vectorizer is designed to interact with Amazon Bedrock API, requiring AWS credentials for authentication. The credentials can be provided directly in the `api_config` dictionary or through environment variables: - AWS_ACCESS_KEY_ID - AWS_SECRET_ACCESS_KEY - AWS_REGION (defaults to us-east-1)

The vectorizer supports synchronous operations with batch processing and preprocessing capabilities.

You can optionally enable caching to improve performance when generating embeddings for repeated inputs.

```
# Basic usage with explicit credentials
vectorizer = BedrockVectorizer(
    model="amazon.titan-embed-text-v2:0",
    api_config={
        "aws_access_key_id": "your_access_key",
        "aws_secret_access_key": "your_secret_key",
        "aws_region": "us-east-1"
    }
)

# With environment variables and caching
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="bedrock_embeddings_cache")

vectorizer = BedrockVectorizer(
    model="amazon.titan-embed-text-v2:0",
    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed("Hello, world!")

# Second call will retrieve from cache
embedding2 = vectorizer.embed("Hello, world!")

# Generate batch embeddings
embeddings = vectorizer.embed_many(["Hello", "World"], batch_size=2)

# Multimodal usage
from pathlib import Path
vectorizer = BedrockVectorizer(
    model="amazon.titan-embed-image-v1:0",
    api_config={
        "aws_access_key_id": "your_access_key",
        "aws_secret_access_key": "your_secret_key",
        "aws_region": "us-east-1"
    }
)
image_embedding = vectorizer.embed(Path("path/to/your/image.jpg"))

# Embedding a list of mixed modalities
embeddings = vectorizer.embed_many(
    ["Hello", "world!", Path("path/to/your/image.jpg")],
    batch_size=2
)
```

Initialize the AWS Bedrock Vectorizer.

Parameters

- **model** (*str*) – The Bedrock model ID to use. Defaults to `amazon.titan-embed-text-v2:0`
- **api_config** (*Optional[Dict[str, str]]*) – AWS credentials and config. Can include: `aws_access_key_id`, `aws_secret_access_key`, `aws_region`. If not provided, will use environment variables.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting `as_buffer=True` in calls to `embed()` and `embed_many()`. Defaults to `'float32'`.
- **cache** (*Optional[EmbeddingsCache]*) – Optional `EmbeddingsCache` instance to cache embeddings for better performance with repeated texts. Defaults to `None`.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ValueError** – If credentials are not provided in config or environment.
- **ImportError** – If `boto3` is not installed.
- **ValueError** – If an invalid dtype is provided.

embed_image(*image_path*, ***kwargs*)

Embed an image (from its path on disk) using a Bedrock multimodal model.

Parameters

image_path (*str*)

Return type

`list[float] | bytes`

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

property type: `str`

Return the type of vectorizer.

CustomVectorizer**Note**

For backwards compatibility, an alias `CustomTextVectorizer` is available in the `redisvl.utils.vectorize.text` module. This alias is deprecated as of version 0.13.0 and will be removed in a future major release.

class CustomVectorizer(*embed*, *embed_many=None*, *aembed=None*, *aembed_many=None*, *dtype='float32'*, *cache=None*)

Bases: `BaseVectorizer`

The `CustomVectorizer` class wraps user-defined embedding methods to create embeddings for data.

This vectorizer is designed to accept a provided callable vectorizer and provides a class definition to allow for compatibility with RedisVL. The vectorizer may support both synchronous and asynchronous operations which allows for batch processing of inputs, but at a minimum only synchronous embedding is required to satisfy the `'embed()'` method.

You can optionally enable caching to improve performance when generating embeddings for repeated inputs.

```
# Basic usage with a custom embedding function
vectorizer = CustomVectorizer(
    embed = my_vectorizer.generate_embedding
)
embedding = vectorizer.embed("Hello, world!")

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="my_embeddings_cache")

vectorizer = CustomVectorizer(
    embed=my_vectorizer.generate_embedding,
    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed("Hello, world!")

# Second call will retrieve from cache
embedding2 = vectorizer.embed("Hello, world!")

# Asynchronous batch embedding of multiple texts
embeddings = await vectorizer.aembed_many(
    ["Hello, world!", "How are you?"],
    batch_size=2
)
```

Initialize the Custom vectorizer.

Parameters

- **embed** (*Callable*) – a Callable function that accepts an object and returns a list of floats.
- **embed_many** (*Optional[Callable]*) – a Callable function that accepts a list of objects and returns a list containing lists of floats. Defaults to None.
- **aembed** (*Optional[Callable]*) – an asynchronous Callable function that accepts a object and returns a lists of floats. Defaults to None.
- **aembed_many** (*Optional[Callable]*) – an asynchronous Callable function that accepts a list of objects and returns a list containing lists of floats. Defaults to None.
- **dtype** (*str*) – the default datatype to use when embedding inputs as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional EmbeddingsCache instance to cache embeddings for better performance with repeated inputs. Defaults to None.

Raises

ValueError – if embedding validation fails.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

property type: `str`

Return the type of vectorizer.

VoyageAIVectorizer

Note

For backwards compatibility, an alias `VoyageAITextVectorizer` is available in the `redisvl.utils.vectorize.text` module. This alias is deprecated as of version 0.13.0 and will be removed in a future major release.

```
class VoyageAIVectorizer(model='voyage-3-large', api_config=None, dtype='float32', cache=None, *,
                        dims=None)
```

Bases: `BaseVectorizer`

The `VoyageAIVectorizer` class utilizes VoyageAI's API to generate embeddings for text and multimodal (text / image / video) data.

This vectorizer is designed to interact with VoyageAI's `/embed`, `/multimodal_embed`, and `/contextualized_embed` APIs. Any model identifier accepted by VoyageAI can be passed via `model` - for example the general-purpose `voyage-4-large` / `voyage-4` / `voyage-4-lite` / `voyage-4-nano` family, domain models such as `voyage-code-4`, contextualized `voyage-context-4` / `voyage-context-3` models, and multimodal `voyage-multimodal-*` models. See <https://docs.voyageai.com/docs/embeddings> for the current catalog.

It requires an API key for authentication. The key can be provided directly in the `api_config` dictionary or through the `VOYAGE_API_KEY` environment variable. User must obtain an API key from VoyageAI's website (<https://dash.voyageai.com/>). Additionally, the `voyageai` python client must be installed with `pip install voyageai`. For image embeddings, the Pillow library must also be installed with `pip install pillow`.

The vectorizer supports both synchronous and asynchronous operations, allows for batch processing of content and flexibility in handling preprocessing tasks.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

```
from redisvl.utils.vectorize import VoyageAIVectorizer

# Basic usage
vectorizer = VoyageAIVectorizer(
    model="voyage-3-large",
    api_config={"api_key": "your-voyageai-api-key"} # OR set VOYAGE_API_KEY in your
    ↪ env
)
query_embedding = vectorizer.embed(
    content="your input query text here",
    input_type="query"
)
doc_embeddings = vectorizer.embed_many(
    contents=["your document text", "more document text"],
    input_type="document"
)

# Contextualized embeddings (voyage-context-* models) - requires voyageai>=0.5.0
# Each input string is treated as its own document (auto-chunked) and
# embedded independently: inputs do not influence one another, which keeps
# the one-embedding-per-input contract and cache determinism intact.
context_vectorizer = VoyageAIVectorizer(
    model="voyage-context-4",
```

(continues on next page)

(continued from previous page)

```

    api_config={"api_key": "your-voyageai-api-key"}
)
context_embeddings = context_vectorizer.embed_many(
    contents=["chunk one", "chunk two", "chunk three"],
    input_type="document"
)
# Retrieval queries use input_type="query"; auto-chunking is a
# document-only feature, so query inputs are embedded as-is.
context_query = context_vectorizer.embed(
    content="your query text here",
    input_type="query"
)

# Multimodal usage - requires Pillow and voyageai>=0.3.6

vectorizer = VoyageAIVectorizer(
    model="voyage-multimodal-3.5",
    api_config={"api_key": "your-voyageai-api-key"} # OR set VOYAGE_API_KEY in your
    → env
)
image_embedding = vectorizer.embed_image(
    "path/to/your/image.jpg",
    input_type="query"
)
video_embedding = vectorizer.embed_video(
    "path/to/your/video.mp4",
    input_type="document"
)

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="voyageai_embeddings_cache")

vectorizer = VoyageAIVectorizer(
    model="voyage-3-large",
    api_config={"api_key": "your-voyageai-api-key"},
    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed(
    content="your input query text here",
    input_type="query"
)

# Second call will retrieve from cache
embedding2 = vectorizer.embed(
    content="your input query text here",
    input_type="query"
)

```

Initialize the VoyageAI vectorizer.

Visit <https://docs.voyageai.com/docs/embeddings> to learn about embeddings and check the available models.

Parameters

- **model** (*str*) – Model to use for embedding. Defaults to “voyage-3-large”.
- **api_config** (*Optional[Dict]*, *optional*) – Dictionary containing the API key. Defaults to None.
- **dtype** (*str*) – the default datatype to use when embedding content as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional *EmbeddingsCache* instance to cache embeddings for better performance with repeated items. Defaults to None.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the *voyageai* library is not installed.
- **ValueError** – If the API key is not provided.

Notes

- **Multimodal models require *voyageai* ≥ 0.3.6 to be installed for video embeddings, as well as *ffmpeg* installed on the system.** Image embeddings require *pillow* to be installed.
- **Contextualized (*voyage-context-**) models require *voyageai* ≥ 0.5.0.** Each input string is sent as its own document with auto-chunking, so inputs are embedded independently (no cross-input contextualization) and the one-embedding-per-input contract and cache determinism are preserved. A document longer than *chunk_size* (32000 tokens) auto-chunks into multiple chunks but only the first chunk’s embedding is kept; the rest are dropped. *truncation* is not forwarded to the contextualized API (it does not accept it), so it is silently ignored for these models.
- **The plain text embedding path (*embed/embed_many* for non-context, non-multimodal models) uses token-aware batching:** inputs are grouped into requests bounded by both the per-model item cap and the model’s per-request token limit, so large inputs are packed efficiently without exceeding VoyageAI’s token budget.

embed_image(*image_path*, ***kwargs*)

Embed an image (from its path on disk) using VoyageAI’s multimodal API. Requires *pillow* to be installed.

Parameters

image_path (*str*)

Return type

list[float] | bytes

embed_video(*video_path*, ***kwargs*)

Embed a video (from its path on disk) using VoyageAI’s multimodal API.

Requires *voyageai* ≥ 0.3.6 to be installed, as well as *ffmpeg* to be installed on the system.

Parameters

video_path (*str*)

Return type

list[float] | bytes

property is_context: bool

Whether a contextualized-embedding model (voyage-context-*) has been configured.

property is_multimodal: bool

Whether a multimodal model has been configured.

model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

property type: str

Return the type of vectorizer.

MistralAITextVectorizer

```
class MistralAITextVectorizer(model='mistral-embed', api_config=None, dtype='float32', cache=None, *,
                             dims=None)
```

Bases: BaseVectorizer

The MistralAITextVectorizer class utilizes MistralAI's API to generate embeddings for text data.

This vectorizer is designed to interact with Mistral's embeddings API, requiring an API key for authentication. The key can be provided directly in the *api_config* dictionary or through the *MISTRAL_API_KEY* environment variable. Users must obtain an API key from Mistral's website (<https://console.mistral.ai/>). Additionally, the *mistralai* python client must be installed with *pip install mistralai*.

The vectorizer supports both synchronous and asynchronous operations, allowing for batch processing of texts and flexibility in handling preprocessing tasks.

You can optionally enable caching to improve performance when generating embeddings for repeated text inputs.

```
# Basic usage
vectorizer = MistralAITextVectorizer(
    model="mistral-embed",
    api_config={"api_key": "your_api_key"} # OR set MISTRAL_API_KEY in your env
)
embedding = vectorizer.embed("Hello, world!")

# With caching enabled
from redisvl.extensions.cache.embeddings import EmbeddingsCache
cache = EmbeddingsCache(name="mistral_embeddings_cache")

vectorizer = MistralAITextVectorizer(
    model="mistral-embed",
    api_config={"api_key": "your_api_key"},
    cache=cache
)

# First call will compute and cache the embedding
embedding1 = vectorizer.embed("Hello, world!")

# Second call will retrieve from cache
embedding2 = vectorizer.embed("Hello, world!")

# Asynchronous batch embedding of multiple texts
```

(continues on next page)

(continued from previous page)

```
embeddings = await vectorizer.aembed_many(
    ["Hello, world!", "How are you?"],
    batch_size=2
)
```

Initialize the MistralAI vectorizer.

Parameters

- **model** (*str*) – Model to use for embedding. Defaults to ‘mistral-embed’.
- **api_config** (*Optional[Dict], optional*) – Dictionary containing the API key. Defaults to None.
- **dtype** (*str*) – the default datatype to use when embedding text as byte arrays. Used when setting *as_buffer=True* in calls to *embed()* and *embed_many()*. Defaults to ‘float32’.
- **cache** (*Optional[EmbeddingsCache]*) – Optional EmbeddingsCache instance to cache embeddings for better performance with repeated texts. Defaults to None.
- **dims** (*Annotated[int | None, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]*)

Raises

- **ImportError** – If the mistralai library is not installed.
- **ValueError** – If the Mistral API key is not provided.
- **ValueError** – If an invalid dtype is provided.

model_config: `ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

property type: `str`

Return the type of vectorizer.

1.4.7 Rerankers

CohereReranker

```
class CohereReranker(model='rerank-english-v3.0', rank_by=None, limit=5, return_score=True,
                    api_config=None)
```

Bases: `BaseReranker`

The CohereReranker class uses Cohere’s API to rerank documents based on an input query.

This reranker is designed to interact with Cohere’s /rerank API, requiring an API key for authentication. The key can be provided directly in the *api_config* dictionary or through the *COHERE_API_KEY* environment variable. User must obtain an API key from Cohere’s website (<https://dashboard.cohere.com/>). Additionally, the *cohere* python client must be installed with *pip install cohere*.

```
from redisvl.utils.rerank import CohereReranker

# set up the Cohere reranker with some configuration
reranker = CohereReranker(rank_by=["content"], limit=2)
```

(continues on next page)

(continued from previous page)

```
# rerank raw search results based on user input/query
results = reranker.rank(
    query="your input query text here",
    docs=[
        {"content": "document 1"},
        {"content": "document 2"},
        {"content": "document 3"}
    ]
)
```

Initialize the CohereReranker with specified model, ranking criteria, and API configuration.

Parameters

- **model** (*str*) – The identifier for the Cohere model used for reranking. Defaults to ‘rerank-english-v3.0’.
- **rank_by** (*Optional[List[str]]*) – Optional list of keys specifying the attributes in the documents that should be considered for ranking. None means ranking will rely on the model’s default behavior.
- **limit** (*int*) – The maximum number of results to return after reranking. Must be a positive integer.
- **return_score** (*bool*) – Whether to return scores alongside the reranked results.
- **api_config** (*Optional[Dict]*, *optional*) – Dictionary containing the API key. Defaults to None.

Raises

- **ImportError** – If the cohere library is not installed.
- **ValueError** – If the API key is not provided.

async arank(*query, docs, **kwargs*)

Rerank documents based on the provided query using the Cohere rerank API.

This method processes the user’s query and the provided documents to rerank them in a manner that is potentially more relevant to the query’s context.

Parameters

- **query** (*str*) – The user’s search query.
- **docs** (*Union[List[Dict[str, Any]], List[str]]*) – The list of documents to be ranked, either as dictionaries or strings.

Returns

The reranked list of documents and optionally associated scores.

Return type

Union[Tuple[Union[List[Dict[str, Any]], List[str]], float], List[Dict[str, Any]]]

model_post_init(*context, /*)

This function is meant to behave like a BaseModel method to initialize private attributes.

It takes context as an argument since that’s what pydantic-core passes when calling it.

Parameters

- **self** (*BaseModel*) – The BaseModel instance.

- **context** (*Any*) – The context.

Return type

None

rank(*query*, *docs*, ***kwargs*)

Rerank documents based on the provided query using the Cohere rerank API.

This method processes the user's query and the provided documents to rerank them in a manner that is potentially more relevant to the query's context.

Parameters

- **query** (*str*) – The user's search query.
- **docs** (*Union[List[Dict[str, Any]], List[str]]*) – The list of documents to be ranked, either as dictionaries or strings.

Returns

The reranked list of documents and optionally associated scores.

Return type

Union[Tuple[Union[List[Dict[str, Any]], List[str]], float], List[Dict[str, Any]]]

model_config: *ClassVar[ConfigDict]* = {}

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

HFCrossEncoderReranker

```
class HFCrossEncoderReranker(model='cross-encoder/ms-marco-MiniLM-L-6-v2', limit=3,
                             return_score=True, *, rank_by=None)
```

Bases: *BaseReranker*

The HFCrossEncoderReranker class uses a cross-encoder models from Hugging Face to rerank documents based on an input query.

This reranker loads a cross-encoder model using the *CrossEncoder* class from the *sentence_transformers* library. It requires the *sentence_transformers* library to be installed.

```
from redisvl.utils.rerank import HFCrossEncoderReranker

# set up the HFCrossEncoderReranker with a specific model
reranker = HFCrossEncoderReranker(model_name="cross-encoder/ms-marco-MiniLM-L-6-v2",
    ↪ limit=3)
# rerank raw search results based on user input/query
results = reranker.rank(
    query="your input query text here",
    docs=[
        {"content": "document 1"},
        {"content": "document 2"},
        {"content": "document 3"}
    ]
)
```

Initialize the HFCrossEncoderReranker with a specified model and ranking criteria.

Parameters

- **model** (*str*) – The name or path of the cross-encoder model to use for reranking. Defaults to ‘cross-encoder/ms-marco-MiniLM-L-6-v2’.
- **limit** (*int*) – The maximum number of results to return after reranking. Must be a positive integer.
- **return_score** (*bool*) – Whether to return scores alongside the reranked results.
- **rank_by** (*list[str] | None*)

async arank(*query, docs, **kwargs*)

Asynchronously rerank documents based on the provided query using the loaded cross-encoder model.

This method processes the user’s query and the provided documents to rerank them in a manner that is potentially more relevant to the query’s context.

Parameters

- **query** (*str*) – The user’s search query.
- **docs** (*Union[List[Dict[str, Any]], List[str]]*) – The list of documents to be ranked, either as dictionaries or strings.

Returns

The reranked list of documents and optionally associated scores.

Return type

Union[Tuple[List[Dict[str, Any]], List[float], List[Dict[str, Any]]]

model_post_init(*context, /*)

This function is meant to behave like a *BaseModel* method to initialize private attributes.

It takes context as an argument since that’s what pydantic-core passes when calling it.

Parameters

- **self** (*BaseModel*) – The *BaseModel* instance.
- **context** (*Any*) – The context.

Return type

None

rank(*query, docs, **kwargs*)

Rerank documents based on the provided query using the loaded cross-encoder model.

This method processes the user’s query and the provided documents to rerank them in a manner that is potentially more relevant to the query’s context.

Parameters

- **query** (*str*) – The user’s search query.
- **docs** (*Union[List[Dict[str, Any]], List[str]]*) – The list of documents to be ranked, either as dictionaries or strings.

Returns

The reranked list of documents and optionally associated scores.

Return type

Union[Tuple[List[Dict[str, Any]], List[float], List[Dict[str, Any]]]

model_config: *ClassVar[ConfigDict]* = {}

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

VoyageAIReranker

class VoyageAIReranker(*model*, *rank_by*=None, *limit*=5, *return_score*=True, *api_config*=None)

Bases: BaseReranker

The VoyageAIReranker class uses VoyageAI's API to rerank documents based on an input query.

This reranker is designed to interact with VoyageAI's /rerank API, requiring an API key for authentication. The key can be provided directly in the *api_config* dictionary or through the *VOYAGE_API_KEY* environment variable. User must obtain an API key from VoyageAI's website (<https://dash.voyageai.com/>). Additionally, the *voyageai* python client must be installed with *pip install voyageai*.

```
from redisvl.utils.rerank import VoyageAIReranker

# set up the VoyageAI reranker with some configuration
reranker = VoyageAIReranker(rank_by=["content"], limit=2)
# rerank raw search results based on user input/query
results = reranker.rank(
    query="your input query text here",
    docs=[
        {"content": "document 1"},
        {"content": "document 2"},
        {"content": "document 3"}
    ]
)
```

Initialize the VoyageAIReranker with specified model, ranking criteria, and API configuration.

Parameters

- **model** (*str*) – The identifier for the VoyageAI model used for reranking.
- **rank_by** (*Optional[List[str]]*) – Optional list of keys specifying the attributes in the documents that should be considered for ranking. None means ranking will rely on the model's default behavior.
- **limit** (*int*) – The maximum number of results to return after reranking. Must be a positive integer.
- **return_score** (*bool*) – Whether to return scores alongside the reranked results.
- **api_config** (*Optional[Dict]*, *optional*) – Dictionary containing the API key. Defaults to None.

Raises

- **ImportError** – If the *voyageai* library is not installed.
- **ValueError** – If the API key is not provided.

async arank(*query*, *docs*, ***kwargs*)

Rerank documents based on the provided query using the VoyageAI rerank API.

This method processes the user's query and the provided documents to rerank them in a manner that is potentially more relevant to the query's context.

Parameters

- **query** (*str*) – The user's search query.
- **docs** (*Union[List[Dict[str, Any]], List[str]]*) – The list of documents to be ranked, either as dictionaries or strings.

Returns

The reranked list of documents and optionally associated scores.

Return type

Union[Tuple[Union[List[Dict[str, Any]], List[str]], float], List[Dict[str, Any]]]

model_post_init(*context*, /)

This function is meant to behave like a BaseModel method to initialize private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** (*BaseModel*) – The BaseModel instance.
- **context** (*Any*) – The context.

Return type

None

rank(*query*, *docs*, ***kwargs*)

Rerank documents based on the provided query using the VoyageAI rerank API.

This method processes the user's query and the provided documents to rerank them in a manner that is potentially more relevant to the query's context.

Parameters

- **query** (*str*) – The user's search query.
- **docs** (*Union[List[Dict[str, Any]], List[str]]*) – The list of documents to be ranked, either as dictionaries or strings.

Returns

The reranked list of documents and optionally associated scores.

Return type

Union[Tuple[Union[List[Dict[str, Any]], List[str]], float], List[Dict[str, Any]]]

model_config: **ClassVar**[**ConfigDict**] = {}

Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.

1.4.8 LLM Cache

SemanticCache

```
class SemanticCache(name='llmcache', distance_threshold=0.1, ttl=None, vectorizer=None,
                    filterable_fields=None, redis_client=None, redis_url='redis://localhost:6379',
                    connection_kwargs=None, overwrite=False, create_index=True, **kwargs)
```

Bases: BaseLLMCache

Semantic Cache for Large Language Models.

Semantic Cache for Large Language Models.

Parameters

- **name** (*str*, *optional*) – The name of the semantic cache search index. Defaults to “llm-cache”.

- **distance_threshold** (*float, optional*) – Semantic distance threshold for the cache in Redis COSINE units [0-2], where lower values indicate stricter matching. Defaults to 0.1.
- **ttl** (*Optional[int], optional*) – The time-to-live for records cached in Redis. Defaults to None.
- **vectorizer** (*Optional[BaseVectorizer], optional*) – The vectorizer for the cache. Defaults to HFTextVectorizer.
- **filterable_fields** (*Optional[List[Dict[str, Any]]]*) – An optional list of RedisVL fields that can be used to customize cache retrieval with filters.
- **redis_client** (*Optional[Redis], optional*) – A redis client connection instance. Defaults to None.
- **redis_url** (*str, optional*) – The redis url. Defaults to redis://localhost:6379.
- **connection_kwargs** (*Dict[str, Any]*) – The connection arguments for the redis client. Defaults to empty {}.
- **overwrite** (*bool*) – Whether or not to force overwrite the schema for the semantic cache index. Defaults to false.
- **create_index** (*bool*) – Whether RedisVL creates and validates the index. When True, the constructor runs FT.INFO to check whether the index exists, compares the live schema against this one, and runs FT.CREATE if it is absent. When False it does none of these and issues no index command at all: the index must already exist with a compatible schema. A live index whose prefix or storage type differs from this schema is not detected and produces empty results rather than an error. Use this when the index is managed externally, or when the credential cannot run FT.INFO. See *Install RedisVL* for the ACL details. Defaults to true.

Raises

- **TypeError** – If an invalid vectorizer is provided.
- **TypeError** – If the TTL value is not an int.
- **ValueError** – If the threshold is not between 0 and 2 (Redis COSINE distance).
- **ValueError** – If existing schema does not match new schema and overwrite is False.
- **ValueError** – If both create_index is False and overwrite is True.

```
from redisvl.extensions.cache.llm import SemanticCache

# RedisVL creates the index if it is missing
cache = SemanticCache(name="llmcache", redis_url="redis://localhost:6379")

# the index is managed externally, or this credential cannot run
# FT.INFO -- assume the index exists and issue no index command
cache = SemanticCache(
    name="llmcache",
    redis_url="redis://localhost:6379",
    create_index=False,
)
```

async acheck(*prompt=None, vector=None, num_results=1, return_fields=None, filter_expression=None, distance_threshold=None*)

Async check the semantic cache for results similar to the specified prompt or vector.

This method searches the cache using vector similarity with either a raw text prompt (converted to a vector) or a provided vector as input. It checks for semantically similar prompts and fetches the cached LLM responses.

Parameters

- **prompt** (*Optional[str], optional*) – The text prompt to search for in the cache.
- **vector** (*Optional[List[float]], optional*) – The vector representation of the prompt to search for in the cache.
- **num_results** (*int, optional*) – The number of cached results to return. Defaults to 1.
- **return_fields** (*Optional[List[str]], optional*) – The fields to include in each returned result. If None, defaults to all available fields in the cached entry.
- **filter_expression** (*Optional[FilterExpression]*) – Optional filter expression that can be used to filter cache results. Defaults to None and the full cache will be searched.
- **distance_threshold** (*Optional[float]*) – The threshold for semantic vector distance.

Returns

A list of dicts containing the requested
return fields for each similar cached response.

Return type

List[Dict[str, Any]]

Raises

- **ValueError** – If neither a *prompt* nor a *vector* is specified.
- **ValueError** – if ‘vector’ has incorrect dimensions.
- **TypeError** – If *return_fields* is not a list when provided.

```
response = await cache.acheck(  
    prompt="What is the capital city of France?"  
)
```

async aclear()

Async delete every cache entry, leaving the index in place.

See [clear\(\)](#) for the caveats, which apply identically here.

Return type

None

async adelete()

Async delete the cache and its index entirely.

Raises

ValueError – If *create_index=False*. See [delete\(\)](#).

Return type

None

async adisconnect()

Asynchronously disconnect from Redis and search index.

Closes all Redis connections and index connections.

async adrop(*ids=None, keys=None*)

Async drop specific entries from the cache by ID or Redis key.

Parameters

- **ids** (*Optional[List[str]]*) – List of entry IDs to remove from the cache. Entry IDs are the unique identifiers without the cache prefix.
- **keys** (*Optional[List[str]]*) – List of full Redis keys to remove from the cache. Keys are the complete Redis keys including the cache prefix.

Return type

None

Note

At least one of ids or keys must be provided.

Raises

ValueError – If neither ids nor keys is provided.

Parameters

- **ids** (*list[str] | None*)
- **keys** (*list[str] | None*)

Return type

None

async aexpire(*key, ttl=None*)

Asynchronously set or refresh the expiration time for a key in the cache.

Parameters

- **key** (*str*) – The Redis key to set the expiration on.
- **ttl** (*Optional[int], optional*) – The time-to-live in seconds. If None, uses the default TTL configured for this cache instance. Defaults to None.

Return type

None

Note

If neither the provided TTL nor the default TTL is set (both are None), this method will have no effect.

async astore(*prompt, response, vector=None, metadata=None, filters=None, ttl=None*)

Async stores the specified key-value pair in the cache along with metadata.

Parameters

- **prompt** (*str*) – The user prompt to cache.
- **response** (*str*) – The LLM response to cache.
- **vector** (*Optional[List[float]], optional*) – The prompt vector to cache. Defaults to None, and the prompt vector is generated on demand.

- **metadata** (*Optional[Dict[str, Any]]*, *optional*) – The optional metadata to cache alongside the prompt and response. Defaults to None.
- **filters** (*Optional[Dict[str, Any]]*) – The optional tag to assign to the cache entry. Defaults to None.
- **ttl** (*Optional[int]*) – The optional TTL override to use on this individual cache entry. Defaults to the global TTL setting.

Returns

The Redis key for the entries added to the semantic cache.

Return type

str

Raises

- **ValueError** – If neither prompt nor vector is specified.
- **ValueError** – if vector has incorrect dimensions.
- **TypeError** – If provided metadata is not a dictionary.

```
key = await cache.astore(
    prompt="What is the capital city of France?",
    response="Paris",
    metadata={"city": "Paris", "country": "France"}
)
```

async aupdate(*key*, ***kwargs*)

Async update specific fields within an existing cache entry. If no fields are passed, then only the document TTL is refreshed.

Parameters

key (str) – the key of the document to update using kwargs.

Raises

- **ValueError** if an incorrect mapping is provided as a kwarg. –
- **TypeError** if metadata is provided and not of type dict. –

Return type

None

```
key = await cache.astore('this is a prompt', 'this is a response')
await cache.aupdate(
    key,
    metadata={"hit_count": 1, "model_name": "Llama-2-7b"}
)
```

check(*prompt=None*, *vector=None*, *num_results=1*, *return_fields=None*, *filter_expression=None*, *distance_threshold=None*)

Checks the semantic cache for results similar to the specified prompt or vector.

This method searches the cache using vector similarity with either a raw text prompt (converted to a vector) or a provided vector as input. It checks for semantically similar prompts and fetches the cached LLM responses.

Parameters

- **prompt** (*Optional[str]*, *optional*) – The text prompt to search for in the cache.

- **vector** (*Optional[List[float]]*, *optional*) – The vector representation of the prompt to search for in the cache.
- **num_results** (*int*, *optional*) – The number of cached results to return. Defaults to 1.
- **return_fields** (*Optional[List[str]]*, *optional*) – The fields to include in each returned result. If None, defaults to all available fields in the cached entry.
- **filter_expression** (*Optional[FilterExpression]*) – Optional filter expression that can be used to filter cache results. Defaults to None and the full cache will be searched.
- **distance_threshold** (*Optional[float]*) – The threshold for semantic vector distance.

Returns

A list of dicts containing the requested
return fields for each similar cached response.

Return type

List[Dict[str, Any]]

Raises

- **ValueError** – If neither a *prompt* nor a *vector* is specified.
- **ValueError** – if ‘vector’ has incorrect dimensions.
- **TypeError** – If *return_fields* is not a list when provided.

```
response = cache.check(
    prompt="What is the capital city of France?"
)
```

clear()

Delete every cache entry, leaving the index in place.

Clears by key prefix, not by index membership, so it removes every key under {name}: and nothing outside it. Available under `create_index=False`; dropping the index is `delete()`.

Warning

Under `create_index=False` the prefix is unverified, so this can delete keys the index never covered and miss entries it does. See [Install RedisVL](#).

Return type

None

delete()

Delete the cache and its index entirely.

Raises

ValueError – If `create_index=False`. Use `clear()` to empty the cache and leave the index standing.

Return type

None

disconnect()

Disconnect from Redis and search index.

Closes all Redis connections and index connections.

drop(*ids=None, keys=None*)

Drop specific entries from the cache by ID or Redis key.

Parameters

- **ids** (*Optional[List[str]]*) – List of entry IDs to remove from the cache. Entry IDs are the unique identifiers without the cache prefix.
- **keys** (*Optional[List[str]]*) – List of full Redis keys to remove from the cache. Keys are the complete Redis keys including the cache prefix.

Return type

None

Note

At least one of ids or keys must be provided.

Raises

ValueError – If neither ids nor keys is provided.

Parameters

- **ids** (*list[str] | None*)
- **keys** (*list[str] | None*)

Return type

None

expire(*key, ttl=None*)

Set or refresh the expiration time for a key in the cache.

Parameters

- **key** (*str*) – The Redis key to set the expiration on.
- **ttl** (*Optional[int], optional*) – The time-to-live in seconds. If None, uses the default TTL configured for this cache instance. Defaults to None.

Return type

None

Note

If neither the provided TTL nor the default TTL is set (both are None), this method will have no effect.

set_threshold(*distance_threshold*)

Sets the semantic distance threshold for the cache.

Parameters

distance_threshold (*float*) – The semantic distance threshold for the cache.

Raises

ValueError – If the threshold is not between 0 and 2 (Redis COSINE distance).

Return type

None

set_ttl(*ttl=None*)

Set the default TTL, in seconds, for entries in the cache.

Parameters

ttl (*Optional[int]*, *optional*) – The optional time-to-live expiration for the cache, in seconds.

Raises

ValueError – If the time-to-live value is not an integer.

Return type

None

store(*prompt, response, vector=None, metadata=None, filters=None, ttl=None*)

Stores the specified key-value pair in the cache along with metadata.

Parameters

- **prompt** (*str*) – The user prompt to cache.
- **response** (*str*) – The LLM response to cache.
- **vector** (*Optional[List[float]]*, *optional*) – The prompt vector to cache. Defaults to None, and the prompt vector is generated on demand.
- **metadata** (*Optional[Dict[str, Any]]*, *optional*) – The optional metadata to cache alongside the prompt and response. Defaults to None.
- **filters** (*Optional[Dict[str, Any]]*) – The optional tag to assign to the cache entry. Defaults to None.
- **ttl** (*Optional[int]*) – The optional TTL override to use on this individual cache entry. Defaults to the global TTL setting.

Returns

The Redis key for the entries added to the semantic cache.

Return type

str

Raises

- **ValueError** – If neither prompt nor vector is specified.
- **ValueError** – if vector has incorrect dimensions.
- **TypeError** – If provided metadata is not a dictionary.

```
key = cache.store(
    prompt="What is the capital city of France?",
    response="Paris",
    metadata={"city": "Paris", "country": "France"}
)
```

update(*key, **kwargs*)

Update specific fields within an existing cache entry. If no fields are passed, then only the document TTL is refreshed.

Parameters

key (*str*) – the key of the document to update using kwargs.

Raises

- **ValueError** if an incorrect mapping is provided as a kwarg. –
- **TypeError** if metadata is provided and not of type dict. –

Return type

None

```
key = cache.store('this is a prompt', 'this is a response')
cache.update(key, metadata={"hit_count": 1, "model_name": "Llama-2-7b"})
```

property aindex: [*AsyncSearchIndex*](#) | None

The underlying AsyncSearchIndex for the cache.

Returns

The async search index.

Return type

[*AsyncSearchIndex*](#)

property distance_threshold: float

The semantic distance threshold for the cache.

Returns

The semantic distance threshold.

Return type

float

property index: [*SearchIndex*](#)

The underlying SearchIndex for the cache.

Returns

The search index.

Return type

[*SearchIndex*](#)

property ttl: int | None

The default TTL, in seconds, for entries in the cache.

LangCacheSemanticCache

```
class LangCacheSemanticCache(name='langcache', server_url='https://aws-us-east-1.langcache.redis.io',
                             cache_id='', api_key='', ttl=None, use_exact_search=True,
                             use_semantic_search=True, distance_scale='normalized', **kwargs)
```

Bases: BaseLLMCache

LLM Cache implementation using the LangCache managed service.

This cache uses the LangCache API service for semantic caching of LLM responses. It requires a LangCache account and API key.

Example

```

from redisvl.extensions.cache.llm import LangCacheSemanticCache

cache = LangCacheSemanticCache(
    name="my_cache",
    server_url="https://api.langcache.com",
    cache_id="your-cache-id",
    api_key="your-api-key",
    ttl=3600
)

# Store a response
cache.store(
    prompt="What is the capital of France?",
    response="Paris"
)

# Check for cached responses
results = cache.check(prompt="What is the capital of France?")

```

Initialize a LangCache semantic cache.

Parameters

- **name** (*str*) – The name of the cache. Defaults to “langcache”.
- **server_url** (*str*) – The LangCache server URL.
- **cache_id** (*str*) – The LangCache cache ID.
- **api_key** (*str*) – The LangCache API key.
- **ttl** (*Optional[int]*) – Time-to-live for cache entries in seconds.
- **use_exact_search** (*bool*) – Whether to use exact matching. Defaults to True.
- **use_semantic_search** (*bool*) – Whether to use semantic search. Defaults to True.
- **distance_scale** (*str*) – Threshold scale for distance_threshold: - “normalized”: 0–1 semantic distance (lower is better) - “redis”: Redis COSINE distance 0–2 (lower is better)

Raises

- **ImportError** – If the langcache package is not installed.
- **ValueError** – If cache_id or api_key is not provided.

async acheck(*prompt=None, vector=None, num_results=1, return_fields=None, filter_expression=None, distance_threshold=None, attributes=None*)

Async check the cache for semantically similar prompts.

Parameters

- **prompt** (*Optional[str]*) – The text prompt to search for.
- **vector** (*Optional[List[float]]*) – Not supported by LangCache API.
- **num_results** (*int*) – Number of results to return. Defaults to 1.
- **return_fields** (*Optional[List[str]]*) – Not used (for compatibility).
- **filter_expression** (*Optional[FilterExpression]*) – Not supported.

- **distance_threshold** (*Optional[float]*) – Maximum distance threshold. Converted to similarity_threshold according to distance_scale: If “redis”, uses norm_cosine_distance(distance_threshold) ([0,2] -> [0,1]). If “normalized”, uses (1.0 - distance_threshold) ([0,1] -> [0,1]).
- **attributes** (*Optional[Dict[str, Any]]*) – LangCache attributes to filter by. Note: Attributes must be pre-configured in your LangCache instance.

Returns

List of matching cache entries.

Return type

List[Dict[str, Any]]

Raises

ValueError – If prompt is not provided.

async aclear()

Async clear the cache of all entries.

This is an alias for adelete() to match the BaseCache interface.

Return type

None

async adelete()

Async delete the entire cache.

This deletes all entries in the cache by calling the flush API.

Return type

None

async adelete_by_attributes(attributes)

Async delete cache entries matching the given attributes.

Parameters

attributes (*Dict[str, Any]*) – Attributes to match for deletion. Cannot be empty.

Returns

Result of the deletion operation.

Return type

Dict[str, Any]

Raises

ValueError – If attributes is an empty dictionary.

async adelete_by_id(entry_id)

Async delete a single cache entry by ID.

Parameters

entry_id (*str*) – The ID of the entry to delete.

Return type

None

async adisconnect()

Async disconnect from Redis.

Return type

None

async aexpire(*key*, *ttl=None*)

Asynchronously set or refresh the expiration time for a key in the cache.

Parameters

- **key** (*str*) – The Redis key to set the expiration on.
- **ttl** (*Optional[int]*, *optional*) – The time-to-live in seconds. If None, uses the default TTL configured for this cache instance. Defaults to None.

Return type

None

Note

If neither the provided TTL nor the default TTL is set (both are None), this method will have no effect.

async astore(*prompt*, *response*, *vector=None*, *metadata=None*, *filters=None*, *ttl=None*)

Async store a prompt-response pair in the cache.

Parameters

- **prompt** (*str*) – The user prompt to cache.
- **response** (*str*) – The LLM response to cache.
- **vector** (*Optional[List[float]]*) – Not supported by LangCache API.
- **metadata** (*Optional[Dict[str, Any]]*) – Optional metadata (stored as attributes).
- **filters** (*Optional[Dict[str, Any]]*) – Not supported.
- **ttl** (*Optional[int]*) – Optional TTL override in seconds.

Returns

The entry ID for the cached entry.

Return type

str

Raises

ValueError – If prompt or response is empty.

async aupdate(*key*, ***kwargs*)

Async update specific fields within an existing cache entry.

Note: LangCache API does not support updating individual entries. This method will raise `NotImplementedError`.

Parameters

- **key** (*str*) – The key of the document to update.
- ****kwargs** – Field-value pairs to update.

Raises

NotImplementedError – LangCache does not support entry updates.

Return type

None

check(*prompt=None, vector=None, num_results=1, return_fields=None, filter_expression=None, distance_threshold=None, attributes=None*)

Check the cache for semantically similar prompts.

Parameters

- **prompt** (*Optional[str]*) – The text prompt to search for.
- **vector** (*Optional[List[float]]*) – Not supported by LangCache API.
- **num_results** (*int*) – Number of results to return. Defaults to 1.
- **return_fields** (*Optional[List[str]]*) – Not used (for compatibility).
- **filter_expression** (*Optional[FilterExpression]*) – Not supported.
- **distance_threshold** (*Optional[float]*) – Maximum distance threshold. Converted to similarity_threshold according to distance_scale: If “redis”, uses norm_cosine_distance(distance_threshold) ([0,2] -> [0,1]). If “normalized”, uses (1.0 - distance_threshold) ([0,1] -> [0,1]).
- **attributes** (*Optional[Dict[str, Any]]*) – LangCache attributes to filter by. Note: Attributes must be pre-configured in your LangCache instance.

Returns

List of matching cache entries.

Return type

List[Dict[str, Any]]

Raises

ValueError – If prompt is not provided.

clear()

Clear the cache of all entries.

This is an alias for delete() to match the BaseCache interface.

Return type

None

delete()

Delete the entire cache.

This deletes all entries in the cache by calling the flush API.

Return type

None

delete_by_attributes(attributes)

Delete cache entries matching the given attributes.

Parameters

attributes (*Dict[str, Any]*) – Attributes to match for deletion. Cannot be empty.

Returns

Result of the deletion operation.

Return type

Dict[str, Any]

Raises

ValueError – If attributes is an empty dictionary.

delete_by_id(*entry_id*)

Delete a single cache entry by ID.

Parameters

entry_id (*str*) – The ID of the entry to delete.

Return type

None

disconnect()

Disconnect from Redis.

Return type

None

expire(*key*, *ttl=None*)

Set or refresh the expiration time for a key in the cache.

Parameters

- **key** (*str*) – The Redis key to set the expiration on.
- **ttl** (*Optional[int]*, *optional*) – The time-to-live in seconds. If None, uses the default TTL configured for this cache instance. Defaults to None.

Return type

None

Note

If neither the provided TTL nor the default TTL is set (both are None), this method will have no effect.

set_ttl(*ttl=None*)

Set the default TTL, in seconds, for entries in the cache.

Parameters

ttl (*Optional[int]*, *optional*) – The optional time-to-live expiration for the cache, in seconds.

Raises

ValueError – If the time-to-live value is not an integer.

Return type

None

store(*prompt*, *response*, *vector=None*, *metadata=None*, *filters=None*, *ttl=None*)

Store a prompt-response pair in the cache.

Parameters

- **prompt** (*str*) – The user prompt to cache.
- **response** (*str*) – The LLM response to cache.
- **vector** (*Optional[List[float]]*) – Not supported by LangCache API.
- **metadata** (*Optional[Dict[str, Any]]*) – Optional metadata (stored as attributes).
- **filters** (*Optional[Dict[str, Any]]*) – Not supported.
- **ttl** (*Optional[int]*) – Optional TTL override in seconds.

Returns

The entry ID for the cached entry.

Return type

str

Raises

ValueError – If prompt or response is empty.

update(key, **kwargs)

Update specific fields within an existing cache entry.

Note: LangCache API does not support updating individual entries. This method will raise `NotImplementedError`.

Parameters

- **key** (str) – The key of the document to update.
- ****kwargs** – Field-value pairs to update.

Raises

NotImplementedError – LangCache does not support entry updates.

Return type

None

property ttl: int | None

The default TTL, in seconds, for entries in the cache.

Cache Schema Classes

CacheEntry

```
class CacheEntry(*, entry_id=None, prompt, response, prompt_vector, inserted_at=<factory>,
                 updated_at=<factory>, metadata=None, filters=None)
```

Bases: BaseModel

A single cache entry in Redis

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **entry_id** (str | None)
- **prompt** (str)
- **response** (str)
- **prompt_vector** (list[float])
- **inserted_at** (float)
- **updated_at** (float)
- **metadata** (dict[str, Any] | None)
- **filters** (dict[str, Any] | None)

entry_id: `str | None`

Cache entry identifier

filters: `dict[str, Any] | None`

Optional filter data stored on the cache entry for customizing retrieval

inserted_at: `float`

Timestamp of when the entry was added to the cache

metadata: `dict[str, Any] | None`

Optional metadata stored on the cache entry

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

prompt: `str`

Input prompt or question cached in Redis

prompt_vector: `list[float]`

Text embedding representation of the prompt

response: `str`

Response or answer to the question, cached in Redis

updated_at: `float`

Timestamp of when the entry was updated in the cache

CacheHit

```
class CacheHit(*, entry_id, prompt, response, vector_distance, inserted_at, updated_at, metadata=None,
               filters=None, **extra_data)
```

Bases: `BaseModel`

A cache hit based on some input query

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **entry_id** (*str*)
- **prompt** (*str*)
- **response** (*str*)
- **vector_distance** (*float*)
- **inserted_at** (*float*)
- **updated_at** (*float*)
- **metadata** (*dict[str, Any] | None*)
- **filters** (*dict[str, Any] | None*)
- **extra_data** (*Any*)

to_dict()

Convert this model to a dictionary, merging filters into the result.

Return type

dict[str, Any]

entry_id: str

Cache entry identifier

filters: dict[str, Any] | None

Optional filter data stored on the cache entry for customizing retrieval

inserted_at: float

Timestamp of when the entry was added to the cache

metadata: dict[str, Any] | None

Optional metadata stored on the cache entry

model_config: ClassVar[ConfigDict] = {'extra': 'allow'}

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

prompt: str

Input prompt or question cached in Redis

response: str

Response or answer to the question, cached in Redis

updated_at: float

Timestamp of when the entry was updated in the cache

vector_distance: float

The semantic distance between the query vector and the stored prompt vector

1.4.9 Embeddings Cache

EmbeddingsCache

```
class EmbeddingsCache(name='embedcache', ttl=None, redis_client=None, async_redis_client=None,
                      redis_url='redis://localhost:6379', connection_kwargs=None)
```

Bases: BaseCache

Embeddings Cache for storing embedding vectors with exact key matching.

Initialize an embeddings cache.

Parameters

- **name** (str) – The name of the cache. Defaults to “embedcache”.
- **ttl** (Optional[int]) – The time-to-live for cached embeddings. Defaults to None.
- **redis_client** (Optional[SyncRedisClient]) – Redis client instance. Defaults to None.
- **redis_url** (str) – Redis URL for connection. Defaults to “redis://localhost:6379”.
- **connection_kwargs** (Dict[str, Any]) – Redis connection arguments. Defaults to {}.
- **async_redis_client** (Redis | RedisCluster | None)

Raises**ValueError** – If vector dimensions are invalid

```
cache = EmbeddingsCache(
    name="my_embeddings_cache",
    ttl=3600, # 1 hour
    redis_url="redis://localhost:6379"
)
```

async aclear()

Asynchronously clear the cache of all keys.

Deletes every Redis key under the cache's prefix (<name>:) with SCAN + DEL. The cache object itself stays usable for future writes.

Note

SCAN is not a point-in-time snapshot, so this is a best-effort sweep rather than an atomic flush:

- Keys written by other clients while the sweep is in progress may or may not be deleted, so the cache is not guaranteed to be empty when this returns. Quiesce writers first if you need that.
- SCAN may return the same key on more than one page. DEL on an already-deleted key is a no-op, so this is harmless.
- Deletion is not atomic across keys. If the call raises partway through, some keys are already gone. The operation is idempotent, so retrying is safe and converges.

Return type

None

async adisconnect()

Async disconnect from Redis.

Return type

None

async adrop(content, model_name)

Async remove an embedding from the cache.

Asynchronously removes an embedding from the cache.

Parameters

- **content** (*bytes* / *str*) – The content that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Return type

None

```
await cache.adrop(
    content="What is machine learning?",
    model_name="text-embedding-ada-002"
)
```

async adrop_by_key(*key*)

Async remove an embedding from the cache by its Redis key.

Asynchronously removes an embedding from the cache by its Redis key.

Parameters

key (*str*) – The full Redis key for the embedding.

Return type

None

```
await cache.adrop_by_key("embedcache:1234567890abcdef")
```

async aexists(*content*, *model_name*)

Async check if an embedding exists.

Asynchronously checks if an embedding exists for the given content and model.

Parameters

- **content** (*bytes* | *str*) – The content that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

True if the embedding exists in the cache, False otherwise.

Return type

bool

```
if await cache.aexists("What is machine learning?", "text-embedding-ada-002"):  
    print("Embedding is in cache")
```

async aexists_by_key(*key*)

Async check if an embedding exists for the given Redis key.

Asynchronously checks if an embedding exists for the given Redis key.

Parameters

key (*str*) – The full Redis key for the embedding.

Returns

True if the embedding exists in the cache, False otherwise.

Return type

bool

```
if await cache.aexists_by_key("embedcache:1234567890abcdef"):  
    print("Embedding is in cache")
```

async aexpire(*key*, *tvl=None*)

Asynchronously set or refresh the expiration time for a key in the cache.

Parameters

- **key** (*str*) – The Redis key to set the expiration on.
- **tvl** (*Optional[int]*, *optional*) – The time-to-live in seconds. If None, uses the default TTL configured for this cache instance. Defaults to None.

Return type

None

Note

If neither the provided TTL nor the default TTL is set (both are None), this method will have no effect.

async aget(*content*, *model_name*)

Async get embedding by content and model name.

Asynchronously retrieves a cached embedding for the given content and model name. If found, refreshes the TTL of the entry.

Parameters

- **content** (*bytes* | *str*) – The content that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

Embedding cache entry or None if not found.

Return type

Optional[Dict[str, Any]]

```
embedding_data = await cache.aget(
    content="What is machine learning?",
    model_name="text-embedding-ada-002"
)
```

async aget_by_key(*key*)

Async get embedding by its full Redis key.

Asynchronously retrieves a cached embedding for the given Redis key. If found, refreshes the TTL of the entry.

Parameters

- **key** (*str*) – The full Redis key for the embedding.

Returns

Embedding cache entry or None if not found.

Return type

Optional[Dict[str, Any]]

```
embedding_data = await cache.aget_by_key("embedcache:1234567890abcdef")
```

async amdrop(*contents*, *model_name*)

Async remove multiple embeddings from the cache by their contents and model name.

Asynchronously removes multiple embeddings in a single operation.

Parameters

- **contents** (*Iterable[bytes* | *str]*) – Iterable of content that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Return type

None

```
# Remove multiple embeddings asynchronously
await cache.amdrops(
    contents=["What is machine learning?", "What is deep learning?"],
    model_name="text-embedding-ada-002"
)
```

async amdrops_by_keys(keys)

Async remove multiple embeddings from the cache by their Redis keys.

Asynchronously removes multiple embeddings in a single operation.

Parameters

keys (*List[str]*) – List of Redis keys to remove.

Return type

None

```
# Remove multiple embeddings asynchronously
await cache.amdrops_by_keys(["embedcache:key1", "embedcache:key2"])
```

async amexists(contents, model_name)

Async check if multiple embeddings exist by their contents and model name.

Asynchronously checks existence of multiple embeddings in a single operation.

Parameters

- **contents** (*Iterable[bytes | str]*) – Iterable of content that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

List of boolean values indicating whether each embedding exists.

Return type

List[bool]

```
# Check if multiple embeddings exist asynchronously
exists_results = await cache.amexists(
    contents=["What is machine learning?", "What is deep learning?"],
    model_name="text-embedding-ada-002"
)
```

async amexists_by_keys(keys)

Async check if multiple embeddings exist by their Redis keys.

Asynchronously checks existence of multiple keys in a single operation.

Parameters

keys (*List[str]*) – List of Redis keys to check.

Returns

List of boolean values indicating whether each key exists. The order matches the input keys order.

Return type

List[bool]

```
# Check if multiple keys exist asynchronously
exists_results = await cache.amexists_by_keys(["embedcache:key1",
↪ "embedcache:key2"])
```

async amget(*contents, model_name*)

Async get multiple embeddings by their contents and model name.

Asynchronously retrieves multiple cached embeddings in a single operation. If found, refreshes the TTL of each entry.

Parameters

- **contents** (*Iterable[bytes | str]*) – Iterable of content that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

List of embedding cache entries or None for contents not found.

Return type

List[Optional[Dict[str, Any]]]

```
# Get multiple embeddings asynchronously
embedding_data = await cache.amget(
    contents=["What is machine learning?", "What is deep learning?"],
    model_name="text-embedding-ada-002"
)
```

async amget_by_keys(*keys*)

Async get multiple embeddings by their Redis keys.

Asynchronously retrieves multiple cached embeddings in a single network roundtrip. If found, refreshes the TTL of each entry.

Parameters

keys (*List[str]*) – List of Redis keys to retrieve.

Returns

List of embedding cache entries or None for keys not found. The order matches the input keys order.

Return type

List[Optional[Dict[str, Any]]]

```
# Get multiple embeddings asynchronously
embedding_data = await cache.amget_by_keys([
    "embedcache:key1",
    "embedcache:key2"
])
```

async amset(*items, ttl=None*)

Async store multiple embeddings in a batch operation.

Each item in the input list should be a dictionary with the following fields: - 'content': The content that was embedded - 'model_name': The name of the embedding model - 'embedding': The embedding vector - 'metadata': Optional metadata to store with the embedding

Parameters

- **items** (*list[dict[str, Any]]*) – List of dictionaries, each containing content, model_name, embedding, and optional metadata.
- **ttl** (*int | None*) – Optional TTL override for these entries.

Returns

List of Redis keys where the embeddings were stored.

Return type

List[str]

Note

The batch is pipelined, not transactional, so on a Redis Cluster it fans out across shards. If it fails partway, some entries will have been written; the operation is idempotent, so simply retry it.

```
# Store multiple embeddings asynchronously
keys = await cache.amset([
    {
        "content": "What is ML?",
        "model_name": "text-embedding-ada-002",
        "embedding": [0.1, 0.2, 0.3],
        "metadata": {"source": "user"}
    },
    {
        "content": "What is AI?",
        "model_name": "text-embedding-ada-002",
        "embedding": [0.4, 0.5, 0.6],
        "metadata": {"source": "docs"}
    }
])
```

async aset(*content, model_name, embedding, metadata=None, ttl=None*)

Async store an embedding with its content and model name.

Asynchronously stores an embedding with its content and model name.

Parameters

- **content** (*bytes | str*) – The content that was embedded.
- **model_name** (*str*) – The name of the embedding model.
- **embedding** (*List[float]*) – The embedding vector to store.
- **metadata** (*Optional[Dict[str, Any]]*) – Optional metadata to store with the embedding.
- **ttl** (*Optional[int]*) – Optional TTL override for this specific entry.

Returns

The Redis key where the embedding was stored.

Return type

str

```
key = await cache.aset(
    content="What is machine learning?",
```

(continues on next page)

(continued from previous page)

```

model_name="text-embedding-ada-002",
embedding=[0.1, 0.2, 0.3, ...],
metadata={"source": "user_query"}
)

```

clear()

Clear the cache of all keys.

Deletes every Redis key under the cache's prefix (<name>:) with SCAN + DEL. The cache object itself stays usable for future writes.

Note

SCAN is not a point-in-time snapshot, so this is a best-effort sweep rather than an atomic flush:

- Keys written by other clients while the sweep is in progress may or may not be deleted, so the cache is not guaranteed to be empty when this returns. Quiesce writers first if you need that.
- SCAN may return the same key on more than one page. DEL on an already-deleted key is a no-op, so this is harmless.
- Deletion is not atomic across keys. If the call raises partway through, some keys are already gone. The operation is idempotent, so retrying is safe and converges.

Return type

None

disconnect()

Disconnect from Redis.

Return type

None

drop(content, model_name)

Remove an embedding from the cache.

Parameters

- **content** (*bytes* / *str*) – The content that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Return type

None

```

cache.drop(
    content="What is machine learning?",
    model_name="text-embedding-ada-002"
)

```

drop_by_key(key)

Remove an embedding from the cache by its Redis key.

Parameters

- **key** (*str*) – The full Redis key for the embedding.

Return type

None

```
cache.drop_by_key("embedcache:1234567890abcdef")
```

exists(*content*, *model_name*)

Check if an embedding exists for the given content and model.

Parameters

- **content** (*bytes* / *str*) – The content that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

True if the embedding exists in the cache, False otherwise.

Return type

bool

```
if cache.exists("What is machine learning?", "text-embedding-ada-002"):  
    print("Embedding is in cache")
```

exists_by_key(*key*)

Check if an embedding exists for the given Redis key.

Parameters

key (*str*) – The full Redis key for the embedding.

Returns

True if the embedding exists in the cache, False otherwise.

Return type

bool

```
if cache.exists_by_key("embedcache:1234567890abcdef"):  
    print("Embedding is in cache")
```

expire(*key*, *ttl=None*)

Set or refresh the expiration time for a key in the cache.

Parameters

- **key** (*str*) – The Redis key to set the expiration on.
- **ttl** (*Optional[int]*, *optional*) – The time-to-live in seconds. If None, uses the default TTL configured for this cache instance. Defaults to None.

Return type

None

Note

If neither the provided TTL nor the default TTL is set (both are None), this method will have no effect.

get(*content*, *model_name*)

Get embedding by content and model name.

Retrieves a cached embedding for the given content and model name. If found, refreshes the TTL of the entry.

Parameters

- **content** (*bytes* / *str*) – The content that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

Embedding cache entry or None if not found.

Return type

Optional[Dict[str, Any]]

```
embedding_data = cache.get(
    content="What is machine learning?",
    model_name="text-embedding-ada-002"
)
```

get_by_key(*key*)

Get embedding by its full Redis key.

Retrieves a cached embedding for the given Redis key. If found, refreshes the TTL of the entry.

Parameters

- **key** (*str*) – The full Redis key for the embedding.

Returns

Embedding cache entry or None if not found.

Return type

Optional[Dict[str, Any]]

```
embedding_data = cache.get_by_key("embedcache:1234567890abcdef")
```

mdrop(*contents*, *model_name*)

Remove multiple embeddings from the cache by their contents and model name.

Efficiently removes multiple embeddings in a single operation.

Parameters

- **contents** (*Iterable[bytes* / *str]*) – Iterable of content that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Return type

None

```
# Remove multiple embeddings
cache.mdrop(
    contents=["What is machine learning?", "What is deep learning?"],
    model_name="text-embedding-ada-002"
)
```

mdrop_by_keys(*keys*)

Remove multiple embeddings from the cache by their Redis keys.

Efficiently removes multiple embeddings in a single operation.

Parameters

keys (*List[str]*) – List of Redis keys to remove.

Return type

None

```
# Remove multiple embeddings
cache.mdrop_by_keys(["embedcache:key1", "embedcache:key2"])
```

mexists(*contents, model_name*)

Check if multiple embeddings exist by their contents and model name.

Efficiently checks existence of multiple embeddings in a single operation.

Parameters

- **contents** (*Iterable[bytes | str]*) – Iterable of content that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

List of boolean values indicating whether each embedding exists.

Return type

List[bool]

```
# Check if multiple embeddings exist
exists_results = cache.mexists(
    contents=["What is machine learning?", "What is deep learning?"],
    model_name="text-embedding-ada-002"
)
```

mexists_by_keys(*keys*)

Check if multiple embeddings exist by their Redis keys.

Efficiently checks existence of multiple keys in a single operation.

Parameters

keys (*List[str]*) – List of Redis keys to check.

Returns

List of boolean values indicating whether each key exists. The order matches the input keys order.

Return type

List[bool]

```
# Check if multiple keys exist
exists_results = cache.mexists_by_keys(["embedcache:key1", "embedcache:key2"])
```

mget(*contents, model_name*)

Get multiple embeddings by their content and model name.

Efficiently retrieves multiple cached embeddings in a single operation. If found, refreshes the TTL of each entry.

Parameters

- **contents** (*Iterable[bytes | str]*) – Iterable of content that was embedded.
- **model_name** (*str*) – The name of the embedding model.

Returns

List of embedding cache entries or None for contents not found.

Return type

List[Optional[Dict[str, Any]]]

```
# Get multiple embeddings
embedding_data = cache.mget(
    contents=["What is machine learning?", "What is deep learning?"],
    model_name="text-embedding-ada-002"
)
```

mget_by_keys(keys)

Get multiple embeddings by their Redis keys.

Efficiently retrieves multiple cached embeddings in a single network roundtrip. If found, refreshes the TTL of each entry.

Parameters

keys (*List[str]*) – List of Redis keys to retrieve.

Returns

List of embedding cache entries or None for keys not found. The order matches the input keys order.

Return type

List[Optional[Dict[str, Any]]]

```
# Get multiple embeddings
embedding_data = cache.mget_by_keys([
    "embedcache:key1",
    "embedcache:key2"
])
```

mset(items, ttl=None)

Store multiple embeddings in a batch operation.

Each item in the input list should be a dictionary with the following fields: - 'content': The input that was embedded - 'model_name': The name of the embedding model - 'embedding': The embedding vector - 'metadata': Optional metadata to store with the embedding

Parameters

- **items** (*list[dict[str, Any]]*) – List of dictionaries, each containing content, model_name, embedding, and optional metadata.
- **ttl** (*int / None*) – Optional TTL override for these entries.

Returns

List of Redis keys where the embeddings were stored.

Return type

List[str]

Note

The batch is pipelined, not transactional, so on a Redis Cluster it fans out across shards. If it fails partway, some entries will have been written; the operation is idempotent, so simply retry it.

```
# Store multiple embeddings
keys = cache.mset([
    {
        "content": "What is ML?",
        "model_name": "text-embedding-ada-002",
        "embedding": [0.1, 0.2, 0.3],
        "metadata": {"source": "user"}
    },
    {
        "content": "What is AI?",
        "model_name": "text-embedding-ada-002",
        "embedding": [0.4, 0.5, 0.6],
        "metadata": {"source": "docs"}
    }
])
```

set(*content*, *model_name*, *embedding*, *metadata=None*, *ttl=None*)

Store an embedding with its content and model name.

Parameters

- **content** (*Union[bytes, str]*) – The content to be embedded.
- **model_name** (*str*) – The name of the embedding model.
- **embedding** (*List[float]*) – The embedding vector to store.
- **metadata** (*Optional[Dict[str, Any]]*) – Optional metadata to store with the embedding.
- **ttl** (*Optional[int]*) – Optional TTL override for this specific entry.

Returns

The Redis key where the embedding was stored.

Return type

str

```
key = cache.set(
    content="What is machine learning?",
    model_name="text-embedding-ada-002",
    embedding=[0.1, 0.2, 0.3, ...],
    metadata={"source": "user_query"}
)
```

set_ttl(*ttl=None*)

Set the default TTL, in seconds, for entries in the cache.

Parameters

- **ttl** (*Optional[int]*, *optional*) – The optional time-to-live expiration for the cache, in seconds.

Raises

ValueError – If the time-to-live value is not an integer.

Return type

None

property ttl: `int | None`

The default TTL, in seconds, for entries in the cache.

1.4.10 LLM Message History

SemanticMessageHistory

```
class SemanticMessageHistory(name, session_tag=None, prefix=None, vectorizer=None,
                             distance_threshold=0.3, redis_client=None, redis_url='redis://localhost:6379',
                             connection_kwargs=None, overwrite=False, create_index=True, **kwargs)
```

Bases: `BaseMessageHistory`

Initialize message history with index

Semantic Message History stores the current and previous user text prompts and LLM responses to allow for enriching future prompts with session context. Message history is stored in individual user or LLM prompts and responses.

Parameters

- **name** (*str*) – The name of the message history index.
- **session_tag** (*Optional[str]*) – Tag to be added to entries to link to a specific conversation session. Defaults to instance ULID.
- **prefix** (*Optional[str]*) – Prefix for the keys for this message data. Defaults to None and will be replaced with the index name.
- **vectorizer** (*Optional[BaseVectorizer]*) – The vectorizer used to create embeddings.
- **distance_threshold** (*float*) – The maximum semantic distance to be included in the context. Defaults to 0.3.
- **redis_client** (*Optional[Redis]*) – A Redis client instance. Defaults to None.
- **redis_url** (*str, optional*) – The redis url. Defaults to `redis://localhost:6379`.
- **connection_kwargs** (*Dict[str, Any]*) – The connection arguments for the redis client. Defaults to empty `{}`.
- **overwrite** (*bool*) – Whether or not to force overwrite the schema for the semantic message index. Defaults to false.
- **create_index** (*bool*) – Whether RedisVL creates and validates the index. When False the constructor issues no index command at all: the index must already exist with a compatible schema, and a live index whose prefix or storage type differs is not detected – which produces empty results rather than an error. See [SemanticCache](#) for a worked example, and [Install RedisVL](#) for the ACL details. Defaults to True.

Raises

ValueError – If both `create_index` is False and `overwrite` is True.

The proposed schema will support a single vector embedding constructed from either the prompt or response in a single string.

add_message(*message, session_tag=None*)

Insert a single prompt or response into the message history. A timestamp is associated with it so that it can be later sorted in sequential ordering after retrieval.

Parameters

- **message** (*Dict[str, str]*) – The user prompt or LLM response.
- **session_tag** (*Optional[str]*) – Tag to be added to entry to link to a specific conversation session. Defaults to instance ULID.

Return type

None

add_messages(*messages, session_tag=None*)

Insert a list of prompts and responses into the session memory. A timestamp is associated with each so that they can be later sorted in sequential ordering after retrieval.

Parameters

- **messages** (*List[Dict[str, str]]*) – The list of user prompts and LLM responses.
- **session_tag** (*Optional[str]*) – Tag to be added to entries to link to a specific conversation session. Defaults to instance ULID.

Return type

None

clear()

Delete every message, leaving the index in place.

Clears by index membership, so it removes the documents the live index covers. Available under `create_index=False`; dropping the index is [delete\(\)](#).

 Warning

Under `create_index=False` the live index is unverified, so if its prefix differs from this instance's it removes documents this instance never wrote. See [Install RedisVL](#).

Return type

None

count(*session_tag=None*)

Count the number of messages in the conversation history.

Parameters

session_tag (*Optional[str]*) – The session tag to filter messages by. If None, counts all messages in the history.

delete()

Remove every message and drop the search index.

Raises

ValueError – If `create_index=False`. Use [clear\(\)](#) to remove the messages and leave the index standing.

Return type

None

drop(*id=None*)

Remove a specific exchange from the message history.

Parameters

id (*Optional[str]*) – The id of the message entry to delete. If None then the last entry is deleted.

Return type

None

get_recent(*top_k=5, as_text=False, raw=False, session_tag=None, role=None*)

Retrieve the recent message history in sequential order.

Parameters

- **top_k** (*int*) – The number of previous exchanges to return. Default is 5.
- **as_text** (*bool*) – Whether to return the conversation as a single string, or list of alternating prompts and responses.
- **raw** (*bool*) – Whether to return the full Redis hash entry or just the prompt and response
- **session_tag** (*Optional[str]*) – Tag of the entries linked to a specific conversation session. Defaults to instance ULID.
- **role** (*Optional[Union[str, List[str]]]*) – Filter messages by role(s). Can be a single role string (“system”, “user”, “llm”, “tool”) or a list of roles. If None, all roles are returned.

Returns**A single string transcription of the session**or list of strings if *as_text* is false.**Return type**

Union[str, List[str]]

Raises**ValueError** – if *top_k* is not an integer greater than or equal to 0, or if *role* contains invalid values.**get_relevant**(*prompt, as_text=False, top_k=5, fall_back=False, session_tag=None, raw=False, distance_threshold=None, role=None*)

Searches the message history for information semantically related to the specified prompt.

This method uses vector similarity search with a text prompt as input. It checks for semantically similar prompts and responses and gets the top k most relevant previous prompts or responses to include as context to the next LLM call.

Parameters

- **prompt** (*str*) – The message text to search for in message history
- **as_text** (*bool*) – Whether to return the prompts and responses as text
- **JSON.** (*or as*)
- **top_k** (*int*) – The number of previous messages to return. Default is 5.
- **session_tag** (*Optional[str]*) – Tag of the entries linked to a specific conversation session. Defaults to instance ULID.
- **distance_threshold** (*Optional[float]*) – The threshold for semantic vector distance.
- **fall_back** (*bool*) – Whether to drop back to recent conversation history if no relevant context is found.
- **raw** (*bool*) – Whether to return the full Redis hash entry or just the message.

- **role** (*Optional[Union[str, List[str]]]*) – Filter messages by role(s). Can be a single role string (“system”, “user”, “llm”, “tool”) or a list of roles. If None, all roles are returned.

Returns

Either a list of strings, or a list of prompts and responses in JSON containing the most relevant.

Return type

Union[List[str], List[Dict[str,str]]

Raises ValueError: if top_k is not an integer greater or equal to 0,
or if role contains invalid values.

store(*prompt, response, session_tag=None*)

Insert a prompt:response pair into the message history. A timestamp is associated with each message so that they can be later sorted in sequential ordering after retrieval.

Parameters

- **prompt** (*str*) – The user prompt to the LLM.
- **response** (*str*) – The corresponding LLM response.
- **session_tag** (*Optional[str]*) – Tag to be added to entries to link to a specific conversation session. Defaults to instance ULID.

Return type

None

property messages: *list[str] | list[dict[str, str]]*

Returns the full message history.

MessageHistory

```
class MessageHistory(name, session_tag=None, prefix=None, redis_client=None,  
                      redis_url='redis://localhost:6379', connection_kwargs=None, create_index=True,  
                      **kwargs)
```

Bases: BaseMessageHistory

Initialize message history

Message History stores the current and previous user text prompts and LLM responses to allow for enriching future prompts with session context. Message history is stored in individual user or LLM prompts and responses.

Parameters

- **name** (*str*) – The name of the message history index.
- **session_tag** (*Optional[str]*) – Tag to be added to entries to link to a specific conversation session. Defaults to instance ULID.
- **prefix** (*Optional[str]*) – Prefix for the keys for this conversation data. Defaults to None and will be replaced with the index name.
- **redis_client** (*Optional[Redis]*) – A Redis client instance. Defaults to None.
- **redis_url** (*str, optional*) – The redis url. Defaults to redis://localhost:6379.
- **connection_kwargs** (*Dict[str, Any]*) – The connection arguments for the redis client. Defaults to empty {}.

- **create_index** (*bool*) – Whether RedisVL creates the index. When False the constructor issues no index command at all: the index must already exist over this name and prefix. This class never validates an existing index’s schema, so nothing further is verified either way – and as elsewhere, a live index whose prefix or storage type differs from this one is not detected and produces empty results rather than an error. See [SemanticCache](#) for a worked example, and [Install RedisVL](#) for the ACL details. Defaults to True.

add_message(*message, session_tag=None*)

Insert a single prompt or response into the message history. A timestamp is associated with it so that it can be later sorted in sequential ordering after retrieval.

Parameters

- **message** (*Dict[str, str]*) – The user prompt or LLM response.
- **session_tag** (*Optional[str]*) – Tag to be added to entries to link to a specific conversation session. Defaults to instance ULID.

Return type

None

add_messages(*messages, session_tag=None*)

Insert a list of prompts and responses into the message history. A timestamp is associated with each so that they can be later sorted in sequential ordering after retrieval.

Parameters

- **messages** (*List[Dict[str, str]]*) – The list of user prompts and LLM responses.
- **session_tag** (*Optional[str]*) – Tag to be added to entries to link to a specific conversation session. Defaults to instance ULID.

Return type

None

clear()

Delete every message, leaving the index in place.

Clears by index membership, so it removes the documents the live index covers. Available under `create_index=False`; dropping the index is [delete\(\)](#).

Warning

Under `create_index=False` the live index is unverified, so if its prefix differs from this instance’s it removes documents this instance never wrote. See [Install RedisVL](#).

Return type

None

count(*session_tag=None*)

Count the number of messages in the conversation history.

Parameters

- **session_tag** (*Optional[str]*) – The session tag to filter messages by. If None, counts all messages in the history.

delete()

Remove every message and drop the search index.

Raises

ValueError – If `create_index=False`. Use `clear()` to remove the messages and leave the index standing.

Return type

None

drop(*id=None*)

Remove a specific exchange from the conversation history.

Parameters

id (*Optional[str]*) – The id of the message entry to delete. If None then the last entry is deleted.

Return type

None

get_recent(*top_k=5, as_text=False, raw=False, session_tag=None, role=None*)

Retrieve the recent message history in sequential order.

Parameters

- **top_k** (*int*) – The number of previous messages to return. Default is 5.
- **as_text** (*bool*) – Whether to return the conversation as a single string, or list of alternating prompts and responses.
- **raw** (*bool*) – Whether to return the full Redis hash entry or just the prompt and response.
- **session_tag** (*Optional[str]*) – Tag of the entries linked to a specific conversation session. Defaults to instance ULID.
- **role** (*Optional[Union[str, List[str]]]*) – Filter messages by role(s). Can be a single role string (“system”, “user”, “llm”, “tool”) or a list of roles. If None, all roles are returned.

Returns

A single string transcription of the messages
or list of strings if `as_text` is false.

Return type

Union[str, List[str]]

Raises

ValueError – if `top_k` is not an integer greater than or equal to 0, or if `role` contains invalid values.

store(*prompt, response, session_tag=None*)

Insert a prompt:response pair into the message history. A timestamp is associated with each exchange so that they can be later sorted in sequential ordering after retrieval.

Parameters

- **prompt** (*str*) – The user prompt to the LLM.
- **response** (*str*) – The corresponding LLM response.
- **session_tag** (*Optional[str]*) – Tag to be added to entries to link to a specific conversation session. Defaults to instance ULID.

Return type

None

property messages: `list[str] | list[dict[str, str]]`

Returns the full message history.

1.4.11 Semantic Router

Semantic Router

```
class SemanticRouter(name, routes, vectorizer=None, routing_config=None, redis_client=None,
                    redis_url='redis://localhost:6379', overwrite=False, connection_kwargs=None,
                    create_index=True)
```

Semantic Router for managing and querying route vectors.

Initialize the SemanticRouter.

Parameters

- **name** (*str*) – The name of the semantic router.
- **routes** (*List[Route]*) – List of Route objects.
- **vectorizer** (*BaseVectorizer, optional*) – The vectorizer used to embed route references. Defaults to default HFTextVectorizer.
- **routing_config** (*RoutingConfig, optional*) – Configuration for routing behavior. Defaults to the default RoutingConfig.
- **redis_client** (*Optional[SyncRedisClient], optional*) – Redis client for connection. Defaults to None.
- **redis_url** (*str, optional*) – The redis url. Defaults to redis://localhost:6379.
- **overwrite** (*bool, optional*) – Whether to overwrite existing index. Defaults to False.
- **connection_kwargs** (*Dict[str, Any]*) – The connection arguments for the redis client. Defaults to empty {}.
- **create_index** (*bool, optional*) – Whether RedisVL creates and validates the index. When False the constructor issues no index command at all and writes nothing: the index must already exist, already hold the reference vectors for **routes**, and already have its stored config, since none of that is written or verified. **routes** must match what is indexed, because each route's distance threshold is applied from this local list – and [add_route\(\)](#) rewrites the stored config from that same list, so attaching with a partial set and then adding a route truncates the config every other client reads. See [SemanticCache](#) for a worked example of the flag, and [Install RedisVL](#) for the ACL details. Defaults to True.

Raises

ValueError – If both `create_index` is False and `overwrite` is True.

add_route(*route*)

Add a new route to the SemanticRouter.

Note that this replaces the router's stored config with this instance's route list, so a router constructed with a subset of the indexed routes will drop the rest from the config that [from_existing\(\)](#) reads.

Embeds the route's references, writes them to the Redis index, appends the route to `self.routes`, and persists the updated router config so the route survives [SemanticRouter.from_existing\(\)](#).

Parameters

route (*Route*) – A fully-formed Route (name, references, distance_threshold, optional meta-data).

Returns

The added route's name.

Return type

str

Raises

ValueError – If a route with this name already exists on the router. Use [add_route_references\(\)](#) to extend an existing route.

add_route_references(*route_name*, *references*)

Add a reference(s) to an existing route.

Parameters

- **router_name** (*str*) – The name of the router.
- **references** (*Union[str, List[str]]*) – The reference or list of references to add.
- **route_name** (*str*)

Returns

The list of added references keys.

Return type

List[str]

clear()

Delete every route reference, leaving the index in place.

Clears by index membership. Available under `create_index=False`; dropping the index is [delete\(\)](#).

 Warning

The stored `route_config` is left as it was, here and on the default path. A separate process calling [from_existing\(\)](#) afterwards will report routes whose reference vectors are gone. [remove_route\(\)](#) keeps the two in step.

Return type

None

delete()

Delete the semantic router index and its persisted route config.

Raises

ValueError – If `create_index=False`. Use [clear\(\)](#) to remove the route references and leave the index standing.

Return type

None

delete_route_references(*route_name=""*, *reference_ids=None*, *keys=None*)

Get references for an existing semantic router route.

Parameters

- **Optional** (*keys*) – The name of the router.
- **Optional** – The reference or list of references to delete.
- **Optional** – List of fully qualified keys (prefix:router:reference_id) to delete.

- **route_name** (*str*)
- **reference_ids** (*list[str] | None*)
- **keys** (*list[str] | None*)

Returns

Number of objects deleted

Return type

int

classmethod **from_dict**(*data, **kwargs*)

Create a SemanticRouter from a dictionary.

Parameters

data (*Dict[str, Any]*) – The dictionary containing the semantic router data.

Returns

The semantic router instance.

Return type

SemanticRouter

Raises

ValueError – If required data is missing or invalid.

```
from redisvl.extensions.router import SemanticRouter
router_data = {
    "name": "example_router",
    "routes": [{"name": "route1", "references": ["ref1"], "distance_threshold": 0.5}],
    "vectorizer": {"type": "openai", "model": "text-embedding-ada-002"},
}
router = SemanticRouter.from_dict(router_data)
```

classmethod **from_existing**(*name, redis_client=None, redis_url='redis://localhost:6379', **kwargs*)

Return SemanticRouter instance from existing index.

Reads the stored route config with JSON.GET, so unlike SearchIndex.from_existing() this needs no FT.INFO. Pass create_index=False to keep it that way through construction, which makes this the way to attach to a router with a credential that cannot run index-metadata commands.

Parameters

- **name** (*str*)
- **redis_client** (*Redis | RedisCluster | None*)
- **redis_url** (*str*)

Return type

SemanticRouter

classmethod **from_yaml**(*file_path, **kwargs*)

Create a SemanticRouter from a YAML file.

Parameters

file_path (*str*) – The path to the YAML file.

Returns

The semantic router instance.

Return type*SemanticRouter***Raises**

- **ValueError** – If the file path is invalid.
- **FileNotFoundError** – If the file does not exist.

```
from redisvl.extensions.router import SemanticRouter
router = SemanticRouter.from_yaml("router.yaml", redis_url="redis://
↪localhost:6379")
```

get(route_name)

Get a route by its name.

Parameters

route_name (*str*) – Name of the route.

Returns

The selected Route object or None if not found.

Return type*Optional[Route]***get_route_references(route_name="", reference_ids=None, keys=None)**

Get references for an existing route route.

Parameters

- **router_name** (*str*) – The name of the router.
- **references** (*Union[str, List[str]]*) – The reference or list of references to add.
- **route_name** (*str*)
- **reference_ids** (*list[str] | None*)
- **keys** (*list[str] | None*)

Returns

Reference objects stored

Return type*List[Dict[str, Any]]***model_post_init(context, /)**

This function is meant to behave like a BaseModel method to initialize private attributes.

It takes context as an argument since that's what pydantic-core passes when calling it.

Parameters

- **self** (*BaseModel*) – The BaseModel instance.
- **context** (*Any*) – The context.

Return type*None***remove_route(route_name)**

Remove a route and all references from the semantic router.

Like [add_route\(\)](#), this replaces the router's stored config with this instance's route list, so removing one route from a router holding only a subset drops the rest from the config [from_existing\(\)](#) reads.

Parameters

route_name (*str*) – Name of the route to remove.

Return type

None

route_many (*statement=None, vector=None, max_k=None, distance_threshold=None, aggregation_method=None*)

Query the semantic router with a given statement or vector for multiple matches.

Parameters

- **statement** (*Optional[str]*) – The input statement to be queried.
- **vector** (*Optional[List[float]]*) – The input vector to be queried.
- **max_k** (*Optional[int]*) – The maximum number of top matches to return.
- **distance_threshold** (*Optional[float]*) – The threshold for semantic distance.
- **aggregation_method** (*Optional[DistanceAggregationMethod]*) – The aggregation method used for vector distances.

Returns

The matching routes and their details.

Return type

List[RouteMatch]

to_dict()

Convert the SemanticRouter instance to a dictionary.

Returns

The dictionary representation of the SemanticRouter.

Return type

Dict[str, Any]

```
from redisvl.extensions.router import SemanticRouter
router = SemanticRouter(name="example_router", routes=[], redis_url="redis://
↪localhost:6379")
router_dict = router.to_dict()
```

to_yaml(file_path, overwrite=True)

Write the semantic router to a YAML file.

Parameters

- **file_path** (*str*) – The path to the YAML file.
- **overwrite** (*bool*) – Whether to overwrite the file if it already exists.

Raises

FileExistsError – If the file already exists and overwrite is False.

Return type

None

```
from redisvl.extensions.router import SemanticRouter
router = SemanticRouter(
    name="example_router",
    routes=[],
```

(continues on next page)

(continued from previous page)

```
    redis_url="redis://localhost:6379"
)
router.to_yaml("router.yaml")
```

update_route_thresholds(*route_thresholds*)

Update the distance thresholds for each route.

Parameters**route_thresholds** (*Dict[str, float]*) – Dictionary of route names and their distance thresholds.**update_routing_config**(*routing_config*)

Update the routing configuration.

Parameters**routing_config** (*RoutingConfig*) – The new routing configuration.**model_config:** *ClassVar[ConfigDict]* = {'arbitrary_types_allowed': True}Configuration for the model, should be a dictionary conforming to *[ConfigDict][pydantic.config.ConfigDict]*.**name:** *str*

The name of the semantic router.

property route_names: *list[str]*

Get the list of route names.

Returns

List of route names.

Return type*List[str]***property route_thresholds:** *dict[str, float | None]*

Get the distance thresholds for each route.

Returns

Dictionary of route names and their distance thresholds.

Return type*Dict[str, float]***routes:** *list[Route]*

List of Route objects.

routing_config: *RoutingConfig*

Configuration for routing behavior.

vectorizer: *BaseVectorizer*

The vectorizer used to embed route references.

Routing Config

class RoutingConfig(*, max_k=1, aggregation_method=DistanceAggregationMethod.avg)

Configuration for routing behavior.

Create a new model by parsing and validating input data from keyword arguments.

Raises [ValidationError][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **max_k** (Annotated[int, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])])
- **aggregation_method** (DistanceAggregationMethod)

max_k: Annotated[int, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0)])]

Aggregation method to use to classify queries.

model_config: ClassVar[ConfigDict] = {'extra': 'ignore'}

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

Route

class Route(*, name, references, metadata={}, distance_threshold=0.5)

Model representing a routing path with associated metadata and thresholds.

Create a new model by parsing and validating input data from keyword arguments.

Raises [ValidationError][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (str)
- **references** (list[str])
- **metadata** (dict[str, Any])
- **distance_threshold** (Annotated[float, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0), Le(le=2)])])

distance_threshold: Annotated[float, FieldInfo(annotation=NoneType, required=True, metadata=[Strict(strict=True), Gt(gt=0), Le(le=2)])]

Distance threshold for matching the route.

metadata: dict[str, Any]

Metadata associated with the route.

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [ConfigDict][pydantic.config.ConfigDict].

name: `str`

The name of the route.

references: `list[str]`

List of reference phrases for the route.

Route Match

class RouteMatch(**, name=None, distance=None*)

Model representing a matched route with distance information.

Create a new model by parsing and validating input data from keyword arguments.

Raises `[ValidationError][pydantic_core.ValidationError]` if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

- **name** (`str | None`)
- **distance** (`float | None`)

distance: `float | None`

The vector distance between the statement and the matched route.

model_config: `ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to `[ConfigDict][pydantic.config.ConfigDict]`.

name: `str | None`

The matched route name.

Distance Aggregation Method

class DistanceAggregationMethod(*value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None*)

Enumeration for distance aggregation methods.

avg = `'avg'`

Compute the average of the vector distances.

min = `'min'`

Compute the minimum of the vector distances.

sum = `'sum'`

Compute the sum of the vector distances.

1.4.12 Command Line Interface

RedisVL provides a command line interface (CLI) called `rvl` for managing vector search indices. The CLI enables you to create, inspect, and delete indices directly from your terminal without writing Python code.

Installation

The `rvl` command is included when you install RedisVL.

```
pip install redisvl
```

Verify the installation by running:

```
rvl version
```

Connection Configuration

The CLI connects to Redis using the following resolution order:

1. The `REDIS_URL` environment variable, if set
2. Explicit connection flags (`--host`, `--port`, `--url`)
3. Default values (`localhost:6379`)

Connection Flags

All commands that interact with Redis accept these optional flags:

Flag	Type	Description	Default
<code>-u, --url</code>	string	Full Redis URL (e.g., <code>redis://localhost:6379</code>)	None
<code>--host</code>	string	Redis server hostname	<code>localhost</code>
<code>-p, --port</code>	integer	Redis server port	<code>6379</code>
<code>--user</code>	string	Redis username for authentication	<code>default</code>
<code>-a, --password</code>	string	Redis password for authentication	Empty
<code>--ssl</code>	flag	Enable SSL/TLS encryption	Disabled

Examples

Connect using environment variable:

```
export REDIS_URL="redis://localhost:6379"
rvl index listall
```

Connect with explicit host and port:

```
rvl index listall --host myredis.example.com --port 6380
```

Connect with authentication and SSL:

```
rvl index listall --user admin --password secret --ssl
```

Getting Help

All commands support the `-h` and `--help` flags to display usage information.

Flag	Description
<code>-h, --help</code>	Display usage information for the command

Examples

```
# Display top-level help
rvl --help

# Display help for a command group
rvl index --help

# Display help for a specific subcommand
rvl index create --help
```

Running `rvl` without any arguments also displays the top-level help message.

Tip

For a hands-on tutorial with practical examples, see the *The RedisVL CLI*.

Commands

rvl version

Display the installed RedisVL version.

Syntax

```
rvl version [OPTIONS]
```

Options

Option	Description
<code>-s, --short</code>	Print only the version number without additional formatting

Examples

```
# Full version output
rvl version

# Version number only
rvl version --short
```

rvl index

Manage vector search indices. This command group provides subcommands for creating, inspecting, listing, and removing indices.

Syntax

```
rvl index <subcommand> [OPTIONS]
```

Subcommands

Subcom- mand	Description
<code>create</code>	Create a new index from a YAML schema file
<code>info</code>	Display detailed information about an index
<code>listall</code>	List all existing indices in the Redis instance
<code>delete</code>	Remove an index while preserving the underlying data
<code>destroy</code>	Remove an index and delete all associated data

rvl index create

Create a new vector search index from a YAML schema definition.

Syntax

```
rvl index create -s <schema_file> [CONNECTION_OPTIONS]
```

Required Options

Option	Description
<code>-s, --schema</code>	Path to the YAML schema file defining the index structure

Example

```
rvl index create -s schema.yaml
```

Schema File Format

The schema file must be valid YAML with the following structure:

```

version: '0.1.0'

index:
  name: my_index
  prefix: doc
  storage_type: hash

fields:
  - name: content
    type: text
  - name: embedding

```

(continues on next page)

(continued from previous page)

```
type: vector
attrs:
  dims: 768
  algorithm: hnsw
  distance_metric: cosine
```

rvl index info

Display detailed information about an existing index, including field definitions and index options.

Syntax

```
rvl index info (-i <index_name> | -s <schema_file>) [OPTIONS]
```

Options

Option	Description
-i, --index	Name of the index to inspect
-s, --schema	Path to the schema file (alternative to specifying index name)

Example

```
rvl index info -i my_index
```

Output

The command displays two tables:

1. **Index Information** containing the index name, storage type, key prefixes, index options, and indexing status
2. **Index Fields** listing each field with its name, attribute, type, and any additional field options

rvl index listall

List all vector search indices in the connected Redis instance.

Syntax

```
rvl index listall [CONNECTION_OPTIONS]
```

Example

```
rvl index listall
```

Output

Returns a numbered list of all index names:

```
Indices:
1. products_index
2. documents_index
3. embeddings_index
```


Note

`listall` uses `FT._LIST`, which Redis tags `@admin` as well as `@search`. An ACL that grants search access and then subtracts `@admin`, such as `+=@search -=@admin`, denies this command with a permission error even though the rest of the CLI works. The migration commands that discover indexes for you are denied by the same ACL: `rvl migrate helper`, `rvl migrate wizard` when no `-i/--index` is given, and `rvl migrate batch-plan --pattern`. See [Install RedisVL](#).

rvl index delete

Remove an index from Redis while preserving the underlying data. Use this when you want to rebuild an index with a different schema without losing your data.

Syntax

```
rvl index delete (-i <index_name> | -s <schema_file>) [CONNECTION_OPTIONS]
```

Options

Option	Description
<code>-i, --index</code>	Name of the index to delete
<code>-s, --schema</code>	Path to the schema file (alternative to specifying index name)

Example

```
rvl index delete -i my_index
```

rvl index destroy

Remove an index and permanently delete all associated data from Redis. This operation cannot be undone.

Syntax

```
rvl index destroy (-i <index_name> | -s <schema_file>) [CONNECTION_OPTIONS]
```

Options

Option	Description
<code>-i, --index</code>	Name of the index to destroy
<code>-s, --schema</code>	Path to the schema file (alternative to specifying index name)

Example

```
rvl index destroy -i my_index
```

Warning

This command permanently deletes both the index and all documents stored with the index prefix. Ensure you have backups before running this command.

rvl stats

Display statistics about an existing index, including document counts, memory usage, and indexing performance metrics.

Syntax

```
rvl stats (-i <index_name> | -s <schema_file>) [OPTIONS]
```

Options

Option	Description
-i, --index	Name of the index to query
-s, --schema	Path to the schema file (alternative to specifying index name)

Example

```
rvl stats -i my_index
```

Statistics Reference

The command returns the following metrics:

Metric	Description
num_docs	Total number of indexed documents
num_terms	Number of distinct terms in text fields
max_doc_id	Highest internal document ID
num_records	Total number of index records
percent_indexed	Percentage of documents fully indexed
hash_indexing_failures	Number of documents that failed to index
number_of_uses	Number of times the index has been queried
bytes_per_record_avg	Average bytes per index record
doc_table_size_mb	Document table size in megabytes
inverted_sz_mb	Inverted index size in megabytes
key_table_size_mb	Key table size in megabytes
offset_bits_per_record_avg	Average offset bits per record
offset_vectors_sz_mb	Offset vectors size in megabytes
offsets_per_term_avg	Average offsets per term
records_per_doc_avg	Average records per document
sortable_values_size_mb	Sortable values size in megabytes
total_indexing_time	Total time spent indexing in milliseconds
total_inverted_index_blocks	Number of inverted index blocks
vector_index_sz_mb	Vector index size in megabytes

rvl migrate

Warning

The index migrator is an **experimental** feature. APIs, CLI commands, and on-disk formats (plans, checkpoints, backups) may change in future releases. Review migration plans carefully before applying to production indexes.

Manage document-preserving index migrations. This command group provides subcommands for planning, executing, and validating schema migrations that preserve existing data.

Syntax

```
rvl migrate <subcommand> [OPTIONS]
```

Subcommands

Subcommand	Description
helper	Show migration guidance and supported capabilities
wizard	Interactively build a migration plan and schema patch
plan	Generate a migration plan from a schema patch or target schema
apply	Execute a reviewed drop/recreate migration plan
estimate	Estimate disk space required for a migration (dry-run)
rollback	Restore original vectors from a backup directory
validate	Validate a completed migration against the live index
batch-plan	Generate a batch migration plan for multiple indexes
batch-apply	Execute a batch migration plan with state tracking
batch-resume	Resume an interrupted batch migration
batch-status	Show status of an in-progress or completed batch migration

rvl migrate plan

Generate a migration plan for a document-preserving drop/recreate migration.

Syntax

```
rvl migrate plan --index <name> (--schema-patch <patch.yaml> | --target-schema <schema.  
→yaml>) [OPTIONS]
```

Required Options

Option	Description
--index, -i	Name of the source index to migrate
--schema-patch	Path to a YAML schema patch file (mutually exclusive with --target-schema)
--target-schema	Path to a full target schema YAML file (mutually exclusive with --schema-patch)

Optional Options

Option	Description
<code>--plan-out</code>	Output path for the migration plan YAML (default: <code>migration_plan.yaml</code>)

Example

```
rvl migrate plan -i my_index --schema-patch changes.yaml --plan-out plan.yaml
```

rvl migrate apply

Execute a reviewed drop/recreate migration plan. Use `--async` for large migrations involving vector quantization.

Warning

Hash vector quantization is unsupported when the same Redis keys are also indexed by another live RediSearch index that expects the old vector datatype. Quantization rewrites vector bytes in the document key itself, so other indexes covering the same key may drop the document or fail to index it. Use an application-level migration with new keys or fields when documents are shared across indexes.

Syntax

```
rvl migrate apply --plan <migration_plan.yaml> --backup-dir <dir> [OPTIONS]
```

Required Options

Option	Description
<code>--plan</code>	Path to the migration plan YAML file
<code>--backup-dir</code>	Required migration backup directory. Vector backup files are written when hash vector bytes are mutated; index-only and JSON migrations validate and record the directory without writing vector backup files.

Optional Options

Option	Description
<code>--async</code>	Run migration asynchronously (recommended for large quantization jobs)
<code>--batch-size</code>	Keys per pipeline batch (default 500)
<code>--workers</code>	Number of parallel workers for quantization (default 1).
<code>--query-check-file</code>	Path to a YAML file with post-migration query checks

Example

```
rvl migrate apply --plan plan.yaml --backup-dir /tmp/backups
rvl migrate apply --plan plan.yaml --async --backup-dir /tmp/backups --workers 4
```

rvl migrate wizard

Interactively build a schema patch and migration plan through a guided wizard.

Syntax

```
rvl migrate wizard [--index <name>] [OPTIONS]
```

Example

```
rvl migrate wizard -i my_index --plan-out plan.yaml
```

rvl migrate rollback

Restore original vector bytes from a retained backup directory. Rollback restores data only; recreate the original index schema separately if the index definition was changed.

Syntax

```
rvl migrate rollback --backup-dir <dir> [--index <name>] [OPTIONS]
```

Required Options

Option	Description
<code>--backup-dir</code>	Directory containing vector backup files from a prior migration

Example

```
rvl migrate rollback --backup-dir /tmp/backups --index my_index
```

rvl migrate batch-plan

Generate a batch plan that applies one shared schema patch to multiple indexes.

Syntax

```
rvl migrate batch-plan --schema-patch <patch.yaml> (--pattern <glob> | --indexes <name1,
↪name2> | --indexes-file <file>) [OPTIONS]
```

rvl migrate batch-apply

Execute a batch migration plan and write checkpoint state for resume.

Syntax

```
rvl migrate batch-apply --plan <batch_plan.yaml> --backup-dir <dir> [OPTIONS]
```

Required Options

Option	Description
<code>--plan</code>	Path to the batch plan YAML file
<code>--backup-dir</code>	Required per-index migration backup directory. Stored in checkpoint state and used for vector backup files when hash vector bytes are mutated.

Example

```
rvl migrate batch-apply --plan batch_plan.yaml --backup-dir /tmp/backups
```

rvl migrate batch-resume

Resume an interrupted batch migration from its checkpoint state.

Syntax

```
rvl migrate batch-resume --state <batch_state.yaml> [--plan <batch_plan.yaml>] [--retry-
↪failed] [--backup-dir <dir>]
```

If `--backup-dir` is omitted, resume uses the backup directory stored in `batch_state.yaml`. Passing a different backup directory for the same checkpoint is rejected.

rvl migrate batch-status

Show status for an in-progress or completed batch migration.

Syntax

```
rvl migrate batch-status --state <batch_state.yaml>
```

Exit Codes

The CLI returns the following exit codes:

Code	Description
0	Command completed successfully
1	Command failed due to missing required arguments or invalid input

Related Resources

- *The RedisVL CLI* for a tutorial-style walkthrough
- *Schema* for YAML schema format details
- *Search Index Classes* for the Python SearchIndex API

1.4.13 Exceptions

RedisVL defines its custom exceptions in `redisvl.exceptions`. Every one of them inherits from `RedisVLError`, so catching that single base class is enough to handle any error the core index and query APIs raise on their own behalf. Catch the more specific subclasses when you want to react differently to, for example, a schema validation failure than to a Redis connection problem. (The MCP integration defines its own `redisvl.mcp.errors.RedisVLMCPError`, which is outside this hierarchy.)

```
Exception
├── RedisVLError
│   ├── RedisSearchError
│   ├── SchemaValidationError
│   ├── QueryValidationError
│   └── RedisModuleVersionError
```

Note

Exceptions raised by the underlying `redis-py` client, such as `redis.exceptions.ConnectionError`, are not part of this hierarchy. Where RedisVL performs an index or search operation on your behalf it wraps those errors in a `RedisSearchError` and chains the original exception, so the underlying cause is still available on `__cause__`. Constructor and argument validation raises standard Python exceptions instead: for example, `VectorQuery(..., ef_runtime=-1)` raises `ValueError` at construction time, before any `try` block around the query run is entered.

When each error is raised

Exception	Raised when	Typical entry points
<code>SchemaValidationError</code>	An object does not match the index schema. Only raised when the index was created with <code>validate_on_load=True</code> .	<code>load()</code>
<code>QueryValidationError</code>	A query is not valid for the index it targets, for example setting <code>ef_runtime</code> on a vector field that uses the <code>flat</code> algorithm.	<code>query()</code>
<code>RedisSearchError</code>	An index or search operation fails, including errors returned by Redis itself.	<code>create()</code> , <code>exists()</code> , <code>delete()</code> , <code>search()</code> , <code>aggregate()</code>
<code>RedisModuleVersionError</code>	The connected Redis or Redis Search version does not support a requested feature, such as an <code>svs-vamana</code> vector field.	<code>create()</code>
<code>RedisVLError</code>	A load operation fails for a reason not covered by a more specific error. Also the base class for everything above.	<code>load()</code>

All of the above apply equally to `SearchIndex` and `AsyncSearchIndex`.

Handling errors

Validating data on load

Schema validation is off by default. Pass `validate_on_load=True` to have RedisVL check each object against the index schema before writing it, and raise `SchemaValidationError` on the first object that does not match.

```
from redisvl.index import SearchIndex
from redisvl.exceptions import SchemaValidationError

index = SearchIndex.from_yaml(
    "schema.yaml",
    redis_url="redis://localhost:6379",
    validate_on_load=True,
)

try:
    index.load(data)
except SchemaValidationError as e:
    # The message identifies the offending object by its position in the
    # input and describes which field failed and why.
    print(f"Invalid record: {e}")
```

The error message reports the index of the object within the batch you passed, so a failure part way through a large load still points at a specific record.

Handling query failures

`QueryValidationError` signals a query that cannot run against this index. It is a programming error rather than a transient one, so it is usually worth failing loudly instead of retrying.

```
from redisvl.query import VectorQuery
from redisvl.exceptions import QueryValidationError

query = VectorQuery(
    vector=[0.1, 0.2, 0.3],
    vector_field_name="embedding",
    return_fields=["title"],
    ef_runtime=50, # only supported by the 'hns' algorithm
)

try:
    results = index.query(query)
except QueryValidationError as e:
    print(f"Query rejected: {e}")
```


Separating configuration problems from Redis problems

`RedisModuleVersionError` is a subclass of `RedisVLError`, not of `RedisSearchError`, so ordering the except clauses lets you distinguish an unsupported feature from a genuine Redis failure.

```
from redisvl.exceptions import RedisModuleVersionError, RedisSearchError

try:
    index.create(overwrite=True)
except RedisModuleVersionError as e:
    # The deployment does not support the requested feature, for example an
    # 'svs-vamana' field on a Redis version without a new enough Redis Search.
    print(f"Unsupported by this Redis deployment: {e}")
except RedisSearchError as e:
    # Something went wrong talking to Redis, or the index definition was
    # rejected. The original redis-py exception is available as e.__cause__.
    print(f"Index creation failed: {e}")
```

Insufficient permissions usually arrive as `RedisSearchError` as well. `create()` checks whether the index already exists before doing anything, so a credential that cannot run `FT.INFO` fails at that check rather than at `FT.CREATE`, with the chained `redis.exceptions.NoPermissionError` on `e.__cause__` naming the denied command. The same applies to an existing index whose key prefix falls outside the credential’s key patterns; for an index that does not exist yet, the check simply reports it as absent and `create()` proceeds.

`listall()` is the exception: it issues `FT._LIST` directly, so a permission failure there raises `redis.exceptions.NoPermissionError` itself rather than a wrapped `RedisSearchError`. See *Install RedisVL* for the ACL categories RedisVL needs.

When the credential genuinely cannot run `FT.INFO`, this error is not something to handle: construct the extension with `create_index=False` instead, which skips the existence check entirely. See *Install RedisVL*.

Telling “the index is missing” apart from other failures

Redis Search reports an absent index as an ordinary error reply rather than a distinct type, and the wording has changed between versions – older releases say `Unknown index name`, Redis 8.6 and earlier say `<name>: no such index`, and Redis 8.8 introduced `SEARCH_INDEX_NOT_FOUND` `Index not found: <name>`. There is no error code to branch on, so code that needs to distinguish “missing” from “something went wrong” has to match the message.

`exists()` already does this for you, which is the reason to prefer it over catching errors from `info()`: it returns `False` only for a recognized missing-index reply and re-raises everything else, so a permission or connection failure is never reported as an absent index.

```
# Prefer this
if not index.exists():
    index.create()

# over inspecting the error yourself, which couples your code to the
# wording of a particular Redis version
try:
    index.info()
except RedisSearchError as e:
    if "no such index" in str(e): # breaks on Redis 8.8
        index.create()
```

When you do need the distinction elsewhere, read `e.__cause__` rather than the `RedisSearchError` message: the wrapper interpolates the index name, so an index whose name happens to contain one of the wordings above would make an unrelated failure look like an absence.

Catching everything

When the calling code only needs to know that the operation failed, catch the base class.

```
from redisvl.exceptions import RedisVLError

try:
    index.load(data)
    results = index.query(query)
except RedisVLError as e:
    logger.error("RedisVL operation failed: %s", e)
    raise
```

Because RedisVL chains the underlying exception when it wraps one, `e.__cause__` still holds the original `redis-py` error where there was one.

Exception classes

RedisVLError

class `RedisVLError`

Bases: `Exception`

Base exception for all RedisVL errors.

RedisSearchError

class `RedisSearchError`

Bases: `RedisVLError`

Error raised for Redis Search specific operations.

SchemaValidationError

class `SchemaValidationError(message, index=None)`

Bases: `RedisVLError`

Error when validating data against a schema.

QueryValidationError

class QueryValidationError

Bases: *RedisVLError*

Error when validating a query.

RedisModuleVersionError

class RedisModuleVersionError

Bases: *RedisVLError*

Error when Redis or module versions are incompatible with requested features.

classmethod *for_svs_vamana*(*min_redis_version*)

Create error for unsupported SVS-VAMANA.

Parameters

min_redis_version (*str*) – Minimum required Redis version

Returns

RedisModuleVersionError with formatted message

Symbols

__eq__() (*Geo method*), 362
 __eq__() (*Num method*), 360
 __eq__() (*Tag method*), 358
 __eq__() (*Text method*), 359
 __eq__() (*Timestamp method*), 364
 __ge__() (*Num method*), 360
 __ge__() (*Timestamp method*), 364
 __gt__() (*Num method*), 361
 __gt__() (*Timestamp method*), 364
 __init__() (*GeoRadius method*), 363
 __le__() (*Num method*), 361
 __le__() (*Timestamp method*), 365
 __lt__() (*Num method*), 361
 __lt__() (*Timestamp method*), 365
 __mod__() (*Tag method*), 358
 __mod__() (*Text method*), 359
 __ne__() (*Geo method*), 363
 __ne__() (*Num method*), 361
 __ne__() (*Tag method*), 358
 __ne__() (*Text method*), 360
 __ne__() (*Timestamp method*), 365
 __str__() (*Geo method*), 363
 __str__() (*Num method*), 362
 __str__() (*Tag method*), 359
 __str__() (*Text method*), 360

A

acheck() (*LangCacheSemanticCache method*), 399
 acheck() (*SemanticCache method*), 391
 aclear() (*EmbeddingsCache method*), 407
 aclear() (*LangCacheSemanticCache method*), 400
 aclear() (*SemanticCache method*), 392
 add_field() (*IndexSchema method*), 265
 add_fields() (*IndexSchema method*), 265
 add_message() (*MessageHistory method*), 423
 add_message() (*SemanticMessageHistory method*), 419
 add_messages() (*MessageHistory method*), 423
 add_messages() (*SemanticMessageHistory method*), 420
 add_route() (*SemanticRouter method*), 425

add_route_references() (*SemanticRouter method*), 426
 add_scores() (*AggregateHybridQuery method*), 330
 add_scores() (*MultiVectorQuery method*), 352
 adelete() (*LangCacheSemanticCache method*), 400
 adelete() (*SemanticCache method*), 392
 adelete_by_attributes() (*LangCacheSemanticCache method*), 400
 adelete_by_id() (*LangCacheSemanticCache method*), 400
 adisconnect() (*EmbeddingsCache method*), 407
 adisconnect() (*LangCacheSemanticCache method*), 400
 adisconnect() (*SemanticCache method*), 392
 adrop() (*EmbeddingsCache method*), 407
 adrop() (*SemanticCache method*), 392
 adrop_by_key() (*EmbeddingsCache method*), 407
 aexists() (*EmbeddingsCache method*), 408
 aexists_by_key() (*EmbeddingsCache method*), 408
 aexpire() (*EmbeddingsCache method*), 408
 aexpire() (*LangCacheSemanticCache method*), 400
 aexpire() (*SemanticCache method*), 393
 aget() (*EmbeddingsCache method*), 409
 aget_by_key() (*EmbeddingsCache method*), 409
 aggregate() (*AsyncSearchIndex method*), 302
 aggregate() (*SearchIndex method*), 291
 AggregateHybridQuery (*class in redisvl.query*), 328
 aindex (*SemanticCache property*), 398
 algorithm (*BaseVectorFieldAttributes attribute*), 275
 algorithm (*FlatVectorFieldAttributes attribute*), 284
 algorithm (*HNSWVectorFieldAttributes attribute*), 277
 algorithm (*SVSConfig attribute*), 287
 algorithm (*SVSVectorFieldAttributes attribute*), 281
 amdrop() (*EmbeddingsCache method*), 409
 amdrop_by_keys() (*EmbeddingsCache method*), 410
 amexists() (*EmbeddingsCache method*), 410
 amexists_by_keys() (*EmbeddingsCache method*), 410
 amget() (*EmbeddingsCache method*), 411
 amget_by_keys() (*EmbeddingsCache method*), 411
 amset() (*EmbeddingsCache method*), 411
 apply() (*AggregateHybridQuery method*), 330
 apply() (*MultiVectorQuery method*), 352

arank() (*CohereReranker* method), 386
 arank() (*HFCrossEncoderReranker* method), 388
 arank() (*VoyageAIReranker* method), 389
 as_redis_field() (*FlatVectorField* method), 283
 as_redis_field() (*GeoField* method), 273
 as_redis_field() (*HNSWVectorField* method), 276
 as_redis_field() (*NumericField* method), 272
 as_redis_field() (*SVSVectorField* method), 280
 as_redis_field() (*TagField* method), 271
 as_redis_field() (*TextField* method), 269
 aset() (*EmbeddingsCache* method), 412
 astore() (*LangCacheSemanticCache* method), 401
 astore() (*SemanticCache* method), 393
 AsyncSearchIndex (class in *redisvl.index*), 301
 attrs (*FlatVectorField* attribute), 283
 attrs (*GeoField* attribute), 274
 attrs (*HNSWVectorField* attribute), 277
 attrs (*NumericField* attribute), 272
 attrs (*SVSVectorField* attribute), 280
 attrs (*TagField* attribute), 271
 attrs (*TextField* attribute), 270
 aupdate() (*LangCacheSemanticCache* method), 401
 aupdate() (*SemanticCache* method), 394
 avg (*DistanceAggregationMethod* attribute), 432
 AzureOpenAITextVectorizer (class in *redisvl.utils.vectorize.text.azureopenai*), 370

B

backend (*GoogleGenAIVectorizer* property), 375
 BaseVectorFieldAttributes (class in *redisvl.schema.fields*), 274
 batch_query() (*AsyncSearchIndex* method), 302
 batch_query() (*SearchIndex* method), 291
 batch_search() (*AsyncSearchIndex* method), 302
 batch_search() (*SearchIndex* method), 292
 batch_size (*VectorQuery* property), 318
 batch_size (*VectorRangeQuery* property), 326
 BedrockVectorizer (class in *redisvl.utils.vectorize.bedrock*), 377
 between() (*Num* method), 362
 between() (*Timestamp* method), 365
 block_size (*FlatVectorFieldAttributes* attribute), 284

C

CacheEntry (class in *redisvl.extensions.cache.llm.schema*), 404
 CacheHit (class in *redisvl.extensions.cache.llm.schema*), 405
 case_sensitive (*TagFieldAttributes* attribute), 272
 check() (*LangCacheSemanticCache* method), 401
 check() (*SemanticCache* method), 394
 clear() (*AsyncSearchIndex* method), 303
 clear() (*EmbeddingsCache* method), 413
 clear() (*LangCacheSemanticCache* method), 402

clear() (*MessageHistory* method), 423
 clear() (*SearchIndex* method), 292
 clear() (*SemanticCache* method), 395
 clear() (*SemanticMessageHistory* method), 420
 clear() (*SemanticRouter* method), 426
 client (*AsyncSearchIndex* property), 310
 client (*SearchIndex* property), 300
 CohereReranker (class in *redisvl.utils.rerank.cohere*), 385
 CohereTextVectorizer (class in *redisvl.utils.vectorize.text.cohere*), 376
 complete (*SearchResults* property), 311
 compression (*SVSConfig* attribute), 287
 compression (*SVSVectorFieldAttributes* attribute), 281
 CompressionAdvisor (class in *redisvl.utils.compression*), 285
 construction_window_size (*SVSConfig* attribute), 288
 construction_window_size (*SVSVectorFieldAttributes* attribute), 281
 count() (*MessageHistory* method), 423
 count() (*SemanticMessageHistory* method), 420
 CountQuery (class in *redisvl.query*), 346
 create() (*AsyncSearchIndex* method), 303
 create() (*SearchIndex* method), 292
 CustomVectorizer (class in *redisvl.utils.vectorize.custom*), 379

D

datatype (*BaseVectorFieldAttributes* attribute), 275
 datatype (*SVSConfig* attribute), 287
 delete() (*AsyncSearchIndex* method), 303
 delete() (*LangCacheSemanticCache* method), 402
 delete() (*MessageHistory* method), 423
 delete() (*SearchIndex* method), 293
 delete() (*SemanticCache* method), 395
 delete() (*SemanticMessageHistory* method), 420
 delete() (*SemanticRouter* method), 426
 delete_by_attributes() (*LangCacheSemanticCache* method), 402
 delete_by_id() (*LangCacheSemanticCache* method), 402
 delete_route_references() (*SemanticRouter* method), 426
 dialect() (*AggregateHybridQuery* method), 330
 dialect() (*CountQuery* method), 347
 dialect() (*FilterQuery* method), 342
 dialect() (*MultiVectorQuery* method), 352
 dialect() (*TextQuery* method), 337
 dialect() (*VectorQuery* method), 313
 dialect() (*VectorRangeQuery* method), 321
 dims (*BaseVectorFieldAttributes* attribute), 275
 disconnect() (*AsyncSearchIndex* method), 304
 disconnect() (*EmbeddingsCache* method), 413

disconnect() (*LangCacheSemanticCache* method), 403
 disconnect() (*SearchIndex* method), 293
 disconnect() (*SemanticCache* method), 395
 disconnect_sync() (*AsyncSearchIndex* method), 304
 distance (*RouteMatch* attribute), 432
 distance_metric (*BaseVectorFieldAttributes* attribute), 275
 distance_threshold (*Route* attribute), 431
 distance_threshold (*SemanticCache* property), 398
 distance_threshold (*VectorRangeQuery* property), 326
 DistanceAggregationMethod (class in *redisvl.extensions.router.schema*), 432
 drop() (*EmbeddingsCache* method), 413
 drop() (*MessageHistory* method), 424
 drop() (*SemanticCache* method), 396
 drop() (*SemanticMessageHistory* method), 420
 drop_by_filter() (*AsyncSearchIndex* method), 304
 drop_by_filter() (*SearchIndex* method), 293
 drop_by_key() (*EmbeddingsCache* method), 413
 drop_documents() (*AsyncSearchIndex* method), 304
 drop_documents() (*SearchIndex* method), 294
 drop_keys() (*AsyncSearchIndex* method), 305
 drop_keys() (*SearchIndex* method), 294

E

ef_construction (*HNSWVectorFieldAttributes* attribute), 277
 ef_runtime (*HNSWVectorFieldAttributes* attribute), 277
 ef_runtime (*VectorQuery* property), 318
 embed_image() (*BedrockVectorizer* method), 379
 embed_image() (*VertexAIVectorizer* method), 373
 embed_image() (*VoyageAIVectorizer* method), 383
 embed_video() (*VertexAIVectorizer* method), 373
 embed_video() (*VoyageAIVectorizer* method), 383
 EmbeddingsCache (class in *redisvl.extensions.cache.embeddings*), 406
 entry_id (*CacheEntry* attribute), 405
 entry_id (*CacheHit* attribute), 406
 epsilon (*HNSWVectorFieldAttributes* attribute), 278
 epsilon (*SVSVectorFieldAttributes* attribute), 281
 epsilon (*VectorRangeQuery* property), 326
 estimate_memory_savings() (*CompressionAdvisor* static method), 285
 exists() (*AsyncSearchIndex* method), 305
 exists() (*EmbeddingsCache* method), 414
 exists() (*SearchIndex* method), 295
 exists_by_key() (*EmbeddingsCache* method), 414
 expander() (*CountQuery* method), 347
 expander() (*FilterQuery* method), 343
 expander() (*TextQuery* method), 338
 expander() (*VectorQuery* method), 314
 expander() (*VectorRangeQuery* method), 321
 expire() (*EmbeddingsCache* method), 414

expire() (*LangCacheSemanticCache* method), 403
 expire() (*SemanticCache* method), 396
 expire_keys() (*AsyncSearchIndex* method), 305
 expire_keys() (*SearchIndex* method), 295

F

fetch() (*AsyncSearchIndex* method), 305
 fetch() (*SearchIndex* method), 295
 field_data (*BaseVectorFieldAttributes* property), 275
 field_names (*IndexSchema* property), 267
 field_weights (*TextQuery* property), 341
 fields (*IndexSchema* attribute), 268
 filter (*CountQuery* property), 350
 filter (*FilterQuery* property), 346
 filter (*TextQuery* property), 341
 filter (*VectorQuery* property), 318
 filter (*VectorRangeQuery* property), 326
 filter() (*AggregateHybridQuery* method), 330
 filter() (*MultiVectorQuery* method), 352
 FilterExpression (class in *redisvl.query.filter*), 357
 FilterQuery (class in *redisvl.query*), 342
 filters (*CacheEntry* attribute), 405
 filters (*CacheHit* attribute), 406
 FlatVectorField (class in *redisvl.schema.fields*), 283
 FlatVectorFieldAttributes (class in *redisvl.schema.fields*), 284
 for_svsvamana() (*RedisModuleVersionError* class method), 447
 from_dict() (*AsyncSearchIndex* class method), 306
 from_dict() (*IndexSchema* class method), 266
 from_dict() (*SearchIndex* class method), 295
 from_dict() (*SemanticRouter* class method), 427
 from_existing() (*AsyncSearchIndex* class method), 306
 from_existing() (*SearchIndex* class method), 296
 from_existing() (*SemanticRouter* class method), 427
 from_yaml() (*AsyncSearchIndex* class method), 306
 from_yaml() (*IndexSchema* class method), 267
 from_yaml() (*SearchIndex* class method), 296
 from_yaml() (*SemanticRouter* class method), 427

G

Geo (class in *redisvl.query.filter*), 362
 GeoField (class in *redisvl.schema.fields*), 273
 GeoFieldAttributes (class in *redisvl.schema.fields*), 274
 GeoRadius (class in *redisvl.query.filter*), 363
 get() (*EmbeddingsCache* method), 414
 get() (*SemanticRouter* method), 428
 get_by_key() (*EmbeddingsCache* method), 415
 get_recent() (*MessageHistory* method), 424
 get_recent() (*SemanticMessageHistory* method), 421
 get_relevant() (*SemanticMessageHistory* method), 421

get_route_references() (*SemanticRouter* method), 428

GoogleGenAIVectorizer (class in *redisvl.utils.vectorize.googlegenai*), 374

graph_max_degree (*SVSConfig* attribute), 288

graph_max_degree (*SVSVectorFieldAttributes* attribute), 281

group_by() (*AggregateHybridQuery* method), 330

group_by() (*MultiVectorQuery* method), 352

H

HFCrossEncoderReranker (class in *redisvl.utils.rerank.hf_cross_encoder*), 387

HFTTextVectorizer (class in *redisvl.utils.vectorize.text.huggingface*), 366

HNSWVectorField (class in *redisvl.schema.fields*), 276

HNSWVectorFieldAttributes (class in *redisvl.schema.fields*), 277

hybrid_policy (*VectorQuery* property), 318

hybrid_policy (*VectorRangeQuery* property), 327

HybridQuery (class in *redisvl.query.hybrid*), 334

I

in_order() (*CountQuery* method), 347

in_order() (*FilterQuery* method), 343

in_order() (*TextQuery* method), 338

in_order() (*VectorQuery* method), 314

in_order() (*VectorRangeQuery* method), 322

index (*IndexSchema* attribute), 268

index (*SemanticCache* property), 398

index_empty (*TagFieldAttributes* attribute), 272

index_empty (*TextFieldAttributes* attribute), 270

index_missing (*BaseVectorFieldAttributes* attribute), 275

IndexSchema (class in *redisvl.schema*), 264

info() (*AsyncSearchIndex* method), 307

info() (*SearchIndex* method), 296

initial_cap (*BaseVectorFieldAttributes* attribute), 275

inserted_at (*CacheEntry* attribute), 405

inserted_at (*CacheHit* attribute), 406

invalidate_sql_schema_cache() (*AsyncSearchIndex* method), 307

invalidate_sql_schema_cache() (*SearchIndex* method), 297

is_context (*VoyageAIVectorizer* property), 383

is_multimodal (*VertexAIVectorizer* property), 373

is_multimodal (*VoyageAIVectorizer* property), 384

K

key() (*AsyncSearchIndex* method), 307

key() (*SearchIndex* method), 297

key_separator (*AsyncSearchIndex* property), 310

key_separator (*SearchIndex* property), 300

L

LangCacheSemanticCache (class in *redisvl.extensions.cache.llm*), 398

language() (*CountQuery* method), 347

language() (*FilterQuery* method), 343

language() (*TextQuery* method), 338

language() (*VectorQuery* method), 314

language() (*VectorRangeQuery* method), 322

limit() (*AggregateHybridQuery* method), 331

limit() (*MultiVectorQuery* method), 353

limit_fields() (*CountQuery* method), 347

limit_fields() (*FilterQuery* method), 343

limit_fields() (*TextQuery* method), 338

limit_fields() (*VectorQuery* method), 314

limit_fields() (*VectorRangeQuery* method), 322

limit_ids() (*CountQuery* method), 347

limit_ids() (*FilterQuery* method), 343

limit_ids() (*TextQuery* method), 338

limit_ids() (*VectorQuery* method), 314

limit_ids() (*VectorRangeQuery* method), 322

listall() (*AsyncSearchIndex* method), 307

listall() (*SearchIndex* method), 297

load() (*AggregateHybridQuery* method), 331

load() (*AsyncSearchIndex* method), 307

load() (*MultiVectorQuery* method), 353

load() (*SearchIndex* method), 297

M

m (*HNSWVectorFieldAttributes* attribute), 278

max_k (*RoutingConfig* attribute), 431

mdrop() (*EmbeddingsCache* method), 415

mdrop_by_keys() (*EmbeddingsCache* method), 415

MessageHistory (class in *redisvl.extensions.message_history.message_history*), 422

messages (*MessageHistory* property), 424

messages (*SemanticMessageHistory* property), 422

metadata (*CacheEntry* attribute), 405

metadata (*CacheHit* attribute), 406

metadata (*Route* attribute), 431

mexists() (*EmbeddingsCache* method), 416

mexists_by_keys() (*EmbeddingsCache* method), 416

mget() (*EmbeddingsCache* method), 416

mget_by_keys() (*EmbeddingsCache* method), 417

min (*DistanceAggregationMethod* attribute), 432

MistralAITextVectorizer (class in *redisvl.utils.vectorize.text.mistral*), 384

model_config (*AzureOpenAITextVectorizer* attribute), 371

model_config (*BaseVectorFieldAttributes* attribute), 275

model_config (*BedrockVectorizer* attribute), 379

model_config (*CacheEntry* attribute), 405

model_config (*CacheHit* attribute), 406

- `model_config` (*CohereReranker* attribute), 387
 - `model_config` (*CohereTextVectorizer* attribute), 377
 - `model_config` (*CustomVectorizer* attribute), 380
 - `model_config` (*FlatVectorField* attribute), 283
 - `model_config` (*FlatVectorFieldAttributes* attribute), 284
 - `model_config` (*GeoField* attribute), 274
 - `model_config` (*GeoFieldAttributes* attribute), 274
 - `model_config` (*GoogleGenAIVectorizer* attribute), 375
 - `model_config` (*HFCrossEncoderReranker* attribute), 388
 - `model_config` (*HFTextVectorizer* attribute), 368
 - `model_config` (*HNSWVectorField* attribute), 277
 - `model_config` (*HNSWVectorFieldAttributes* attribute), 278
 - `model_config` (*IndexSchema* attribute), 268
 - `model_config` (*MistralAITextVectorizer* attribute), 385
 - `model_config` (*NumericField* attribute), 272
 - `model_config` (*NumericFieldAttributes* attribute), 273
 - `model_config` (*OpenAITextVectorizer* attribute), 369
 - `model_config` (*Route* attribute), 431
 - `model_config` (*RouteMatch* attribute), 432
 - `model_config` (*RoutingConfig* attribute), 431
 - `model_config` (*SemanticRouter* attribute), 430
 - `model_config` (*SVSConfig* attribute), 288
 - `model_config` (*SVSVectorField* attribute), 280
 - `model_config` (*SVSVectorFieldAttributes* attribute), 281
 - `model_config` (*TagField* attribute), 271
 - `model_config` (*TagFieldAttributes* attribute), 272
 - `model_config` (*TextField* attribute), 270
 - `model_config` (*TextFieldAttributes* attribute), 270
 - `model_config` (*Vector* attribute), 312
 - `model_config` (*VertexAIVectorizer* attribute), 373
 - `model_config` (*VoyageAIReranker* attribute), 390
 - `model_config` (*VoyageAIVectorizer* attribute), 384
 - `model_post_init()` (*CohereReranker* method), 386
 - `model_post_init()` (*HFCrossEncoderReranker* method), 388
 - `model_post_init()` (*HFTextVectorizer* method), 368
 - `model_post_init()` (*SemanticRouter* method), 428
 - `model_post_init()` (*VoyageAIReranker* method), 390
 - `mset()` (*EmbeddingsCache* method), 417
 - `MultiVectorQuery` (class in *redisvl.query*), 351
- ## N
- `name` (*AsyncSearchIndex* property), 310
 - `name` (*Route* attribute), 431
 - `name` (*RouteMatch* attribute), 432
 - `name` (*SearchIndex* property), 301
 - `name` (*SemanticRouter* attribute), 430
 - `no_content()` (*CountQuery* method), 348
 - `no_content()` (*FilterQuery* method), 343
 - `no_content()` (*TextQuery* method), 338
 - `no_content()` (*VectorQuery* method), 314
 - `no_content()` (*VectorRangeQuery* method), 322
 - `no_stem` (*TextFieldAttributes* attribute), 270
 - `no_stopwords()` (*CountQuery* method), 348
 - `no_stopwords()` (*FilterQuery* method), 343
 - `no_stopwords()` (*TextQuery* method), 338
 - `no_stopwords()` (*VectorQuery* method), 314
 - `no_stopwords()` (*VectorRangeQuery* method), 322
 - `Num` (class in *redisvl.query.filter*), 360
 - `NumericField` (class in *redisvl.schema.fields*), 272
 - `NumericFieldAttributes` (class in *redisvl.schema.fields*), 273
- ## O
- `OpenAITextVectorizer` (class in *redisvl.utils.vectorize.text.openai*), 368
- ## P
- `paginate()` (*AsyncSearchIndex* method), 308
 - `paginate()` (*SearchIndex* method), 298
 - `paging()` (*CountQuery* method), 348
 - `paging()` (*FilterQuery* method), 344
 - `paging()` (*TextQuery* method), 338
 - `paging()` (*VectorQuery* method), 314
 - `paging()` (*VectorRangeQuery* method), 322
 - `params` (*AggregateHybridQuery* property), 332
 - `params` (*CountQuery* property), 350
 - `params` (*FilterQuery* property), 346
 - `params` (*MultiVectorQuery* property), 354
 - `params` (*TextQuery* property), 341
 - `params` (*VectorQuery* property), 318
 - `params` (*VectorRangeQuery* property), 327
 - `phonetic_matcher` (*TextFieldAttributes* attribute), 270
 - `prefix` (*AsyncSearchIndex* property), 310
 - `prefix` (*SearchIndex* property), 301
 - `prefixes` (*AsyncSearchIndex* property), 310
 - `prefixes` (*SearchIndex* property), 301
 - `prompt` (*CacheEntry* attribute), 405
 - `prompt` (*CacheHit* attribute), 406
 - `prompt_vector` (*CacheEntry* attribute), 405
- ## Q
- `query` (*CountQuery* property), 350
 - `query` (*FilterQuery* property), 346
 - `query` (*TextQuery* property), 341
 - `query` (*VectorQuery* property), 319
 - `query` (*VectorRangeQuery* property), 327
 - `query()` (*AsyncSearchIndex* method), 309
 - `query()` (*SearchIndex* method), 299
 - `query_string()` (*CountQuery* method), 348
 - `query_string()` (*FilterQuery* method), 344
 - `query_string()` (*TextQuery* method), 339
 - `query_string()` (*VectorQuery* method), 315
 - `query_string()` (*VectorRangeQuery* method), 323
 - `QueryValidationError` (class in *redisvl.exceptions*), 447

R

rank() (*CohereReranker* method), 387
 rank() (*HFCrossEncoderReranker* method), 388
 rank() (*VoyageAIReranker* method), 390
 recommend() (*CompressionAdvisor* static method), 286
 redis_query_string() (*SQLQuery* method), 355
 RedisModuleVersionError (class in *redisvl.exceptions*), 447
 RedisSearchError (class in *redisvl.exceptions*), 446
 RedisVLError (class in *redisvl.exceptions*), 446
 reduce (*SVSConfig* attribute), 287
 reduce (*SVSVectorFieldAttributes* attribute), 281
 references (*Route* attribute), 432
 remove_field() (*IndexSchema* method), 267
 remove_route() (*SemanticRouter* method), 428
 response (*CacheEntry* attribute), 405
 response (*CacheHit* attribute), 406
 return_fields() (*CountQuery* method), 348
 return_fields() (*FilterQuery* method), 344
 return_fields() (*TextQuery* method), 339
 return_fields() (*VectorQuery* method), 315
 return_fields() (*VectorRangeQuery* method), 323
 Route (class in *redisvl.extensions.router*), 431
 route_many() (*SemanticRouter* method), 429
 route_names (*SemanticRouter* property), 430
 route_thresholds (*SemanticRouter* property), 430
 RouteMatch (class in *redisvl.extensions.router.schema*), 432
 routes (*SemanticRouter* attribute), 430
 routing_config (*SemanticRouter* attribute), 430
 RoutingConfig (class in *redisvl.extensions.router*), 431

S

SchemaValidationError (class in *redisvl.exceptions*), 446
 scorer() (*AggregateHybridQuery* method), 332
 scorer() (*CountQuery* method), 348
 scorer() (*FilterQuery* method), 344
 scorer() (*MultiVectorQuery* method), 353
 scorer() (*TextQuery* method), 339
 scorer() (*VectorQuery* method), 315
 scorer() (*VectorRangeQuery* method), 323
 search() (*AsyncSearchIndex* method), 310
 search() (*SearchIndex* method), 299
 search_buffer_capacity (*VectorQuery* property), 319
 search_buffer_capacity (*VectorRangeQuery* property), 327
 search_window_size (*SVSConfig* attribute), 288
 search_window_size (*SVSVectorFieldAttributes* attribute), 281
 search_window_size (*VectorQuery* property), 319
 search_window_size (*VectorRangeQuery* property), 327

SearchIndex (class in *redisvl.index*), 290
 SearchResults (class in *redisvl.index*), 311
 SemanticCache (class in *redisvl.extensions.cache.llm*), 390
 SemanticMessageHistory (class in *redisvl.extensions.message_history.semantic_history*), 419
 SemanticRouter (class in *redisvl.extensions.router*), 425
 separator (*TagFieldAttributes* attribute), 272
 set() (*EmbeddingsCache* method), 418
 set_batch_size() (*VectorQuery* method), 315
 set_batch_size() (*VectorRangeQuery* method), 323
 set_distance_threshold() (*VectorRangeQuery* method), 323
 set_ef_runtime() (*VectorQuery* method), 315
 set_epsilon() (*VectorRangeQuery* method), 324
 set_field_weights() (*TextQuery* method), 339
 set_filter() (*CountQuery* method), 349
 set_filter() (*FilterQuery* method), 344
 set_filter() (*TextQuery* method), 339
 set_filter() (*VectorQuery* method), 316
 set_filter() (*VectorRangeQuery* method), 324
 set_hybrid_policy() (*VectorQuery* method), 316
 set_hybrid_policy() (*VectorRangeQuery* method), 324
 set_search_buffer_capacity() (*VectorQuery* method), 316
 set_search_buffer_capacity() (*VectorRangeQuery* method), 324
 set_search_window_size() (*VectorQuery* method), 316
 set_search_window_size() (*VectorRangeQuery* method), 324
 set_text_weights() (*AggregateHybridQuery* method), 332
 set_text_weights() (*TextQuery* method), 340
 set_threshold() (*SemanticCache* method), 396
 set_ttl() (*EmbeddingsCache* method), 418
 set_ttl() (*LangCacheSemanticCache* method), 403
 set_ttl() (*SemanticCache* method), 397
 set_use_search_history() (*VectorQuery* method), 316
 set_use_search_history() (*VectorRangeQuery* method), 324
 slop() (*CountQuery* method), 349
 slop() (*FilterQuery* method), 345
 slop() (*TextQuery* method), 340
 slop() (*VectorQuery* method), 317
 slop() (*VectorRangeQuery* method), 325
 sort_by() (*AggregateHybridQuery* method), 332
 sort_by() (*CountQuery* method), 349
 sort_by() (*FilterQuery* method), 345
 sort_by() (*MultiVectorQuery* method), 354

sort_by() (*TextQuery* method), 340
 sort_by() (*VectorQuery* method), 317
 sort_by() (*VectorRangeQuery* method), 325
 SQLQuery (class in *redisvl.query*), 354
 stopwords (*AggregateHybridQuery* property), 333
 storage_type (*AsyncSearchIndex* property), 310
 storage_type (*SearchIndex* property), 301
 store() (*LangCacheSemanticCache* method), 403
 store() (*MessageHistory* method), 424
 store() (*SemanticCache* method), 397
 store() (*SemanticMessageHistory* method), 422
 sum (*DistanceAggregationMethod* attribute), 432
 SVSConfig (class in *redisvl.utils.compression*), 287
 SVSVectorField (class in *redisvl.schema.fields*), 279
 SVSVectorFieldAttributes (class in *redisvl.schema.fields*), 280

T

Tag (class in *redisvl.query.filter*), 358
 TagField (class in *redisvl.schema.fields*), 271
 TagFieldAttributes (class in *redisvl.schema.fields*), 271
 Text (class in *redisvl.query.filter*), 359
 text_field_name (*TextQuery* property), 341
 text_weights (*AggregateHybridQuery* property), 333
 text_weights (*TextQuery* property), 342
 TextField (class in *redisvl.schema.fields*), 269
 TextFieldAttributes (class in *redisvl.schema.fields*), 270
 TextQuery (class in *redisvl.query*), 336
 timeout() (*CountQuery* method), 350
 timeout() (*FilterQuery* method), 345
 timeout() (*TextQuery* method), 341
 timeout() (*VectorQuery* method), 317
 timeout() (*VectorRangeQuery* method), 326
 Timestamp (class in *redisvl.query.filter*), 364
 to_dict() (*CacheHit* method), 406
 to_dict() (*IndexSchema* method), 267
 to_dict() (*SemanticRouter* method), 429
 to_yaml() (*IndexSchema* method), 267
 to_yaml() (*SemanticRouter* method), 429
 training_threshold (*SVSVectorFieldAttributes* attribute), 281
 ttl (*EmbeddingsCache* property), 418
 ttl (*LangCacheSemanticCache* property), 404
 ttl (*SemanticCache* property), 398
 type (*AzureOpenAITextVectorizer* property), 371
 type (*BedrockVectorizer* property), 379
 type (*CohereTextVectorizer* property), 377
 type (*CustomVectorizer* property), 380
 type (*FlatVectorField* attribute), 284
 type (*GeoField* attribute), 274
 type (*GoogleGenAIVectorizer* property), 375
 type (*HFTTextVectorizer* property), 368

type (*HNSWVectorField* attribute), 277
 type (*MistralAITextVectorizer* property), 385
 type (*NumericField* attribute), 273
 type (*OpenAITextVectorizer* property), 369
 type (*SVSVectorField* attribute), 280
 type (*TagField* attribute), 271
 type (*TextField* attribute), 270
 type (*VertexAIVectorizer* property), 373
 type (*VoyageAIVectorizer* property), 384

U

unf (*NumericFieldAttributes* attribute), 273
 unf (*TextFieldAttributes* attribute), 270
 update() (*LangCacheSemanticCache* method), 404
 update() (*SemanticCache* method), 397
 update_by_filter() (*AsyncSearchIndex* method), 310
 update_by_filter() (*SearchIndex* method), 299
 update_route_thresholds() (*SemanticRouter* method), 430
 update_routing_config() (*SemanticRouter* method), 430
 updated_at (*CacheEntry* attribute), 405
 updated_at (*CacheHit* attribute), 406
 uppercase_strings() (*BaseVectorFieldAttributes* class method), 275
 use_search_history (*VectorQuery* property), 319
 use_search_history (*VectorRangeQuery* property), 327

V

validate_svs_params() (*SVSVectorFieldAttributes* method), 281
 validate_vector() (*Vector* method), 312
 Vector (class in *redisvl.query*), 311
 vector_distance (*CacheHit* attribute), 406
 vectorizer (*SemanticRouter* attribute), 430
 VectorQuery (class in *redisvl.query*), 312
 VectorRangeQuery (class in *redisvl.query*), 320
 verbatim() (*CountQuery* method), 350
 verbatim() (*FilterQuery* method), 346
 verbatim() (*TextQuery* method), 341
 verbatim() (*VectorQuery* method), 318
 verbatim() (*VectorRangeQuery* method), 326
 version (*IndexSchema* attribute), 268
 VertexAIVectorizer (class in *redisvl.utils.vectorize.vertexai*), 371
 VoyageAIReranker (class in *redisvl.utils.rerank.voyageai*), 389
 VoyageAIVectorizer (class in *redisvl.utils.vectorize.voyageai*), 381

W

weight (*TextFieldAttributes* attribute), 270
 with_payloads() (*CountQuery* method), 350

`with_payloads()` (*FilterQuery* method), 346
`with_payloads()` (*TextQuery* method), 341
`with_payloads()` (*VectorQuery* method), 318
`with_payloads()` (*VectorRangeQuery* method), 326
`with_schema()` (*AggregateHybridQuery* method), 332
`with_schema()` (*MultiVectorQuery* method), 354
`with_scores()` (*CountQuery* method), 350
`with_scores()` (*FilterQuery* method), 346
`with_scores()` (*TextQuery* method), 341
`with_scores()` (*VectorQuery* method), 318
`with_scores()` (*VectorRangeQuery* method), 326
`withsuffixtrie` (*TagFieldAttributes* attribute), 272
`withsuffixtrie` (*TextFieldAttributes* attribute), 271