Rockset Hybrid Search Architecture

All vector search is hybrid search

All vector search is becoming hybrid search as it drives the most relevant, real-time application experiences. Hybrid search involves incorporating vector search and text search as well as metadata filtering, all in a single query. Hybrid search is used in search, recommendations and retrieval augmented generation (RAG) applications.

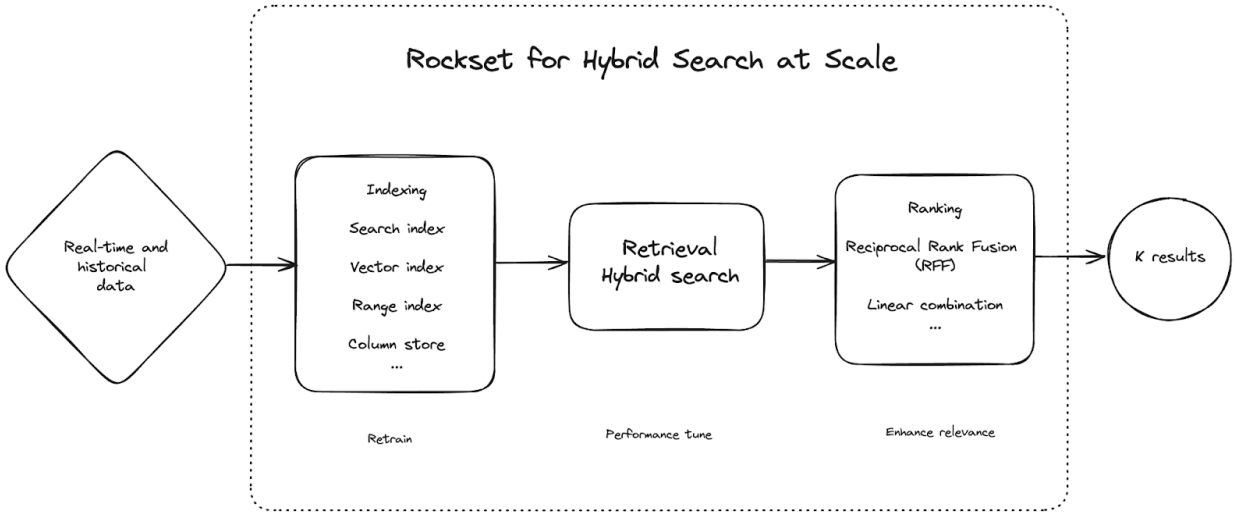
Vector search is a way to find related text, images or videos that have similar characteristics using machine learning models. While vector embeddings using large language models (LLMs) have rapidly advanced and become widely accessible, the generalizable nature of these models means that they are rarely used in isolation in production applications. There are several reasons for this including:

- **Domain awareness:** Generalizable models have limitations around domain awareness; they are not familiar with terminology used by an internal team or industry. Text search is helpful at finding specific terminology.
- **Filters:** Similarity search is designed to find items that are contextually alike. It is not designed for exact filters such as a delivery application that needs to find restaurants that only have 4+ stars and are within a 10 mile radius. In this example, metadata can be used to filter application results.
- **Referenceability:** While natural language responses from LLMs can provide helpful information and summarization, we see users verify the results or dig deeper using source material.
- **Hallucinations:** It's well known that LLMs are prone to hallucinations and grounding them with contextual information can improve accuracy.
- **Permissioning:** Enterprises use permissions to ensure that answers are privacy-aware and secure by only giving the LLM access to data accessible to users.

Given these reasons, we see vector search intertwined with text search, relational search and geospatial search to drive the most relevant results. There are several investments that vector databases need to make to design for hybrid search:

- **Fast complex search:** Ability to support complex search across vectors, text, geo, JSON and other data types to provide necessary context.
- **Support for data and index changes:** The processing of data, retraining of indexes, and performance tuning of search all impact relevancy. Users will iterate on indexing algorithms and model inputs over time, needing to quickly and efficiently make changes.
- **Rankings:** With a hybrid approach, ranking becomes crucial to AI applications. Incorporating a myriad of ranking algorithms and iterating on rank weighting enables users to quickly enhance application relevance.
- **Real-time updates:** Many applications serve real-time data at scale to enhance relevance. The ability to update and delete any data, including vectors, in milliseconds while avoiding high reindexing costs makes hybrid search economical.

Given the considerations above, Rockset is designed and optimized to ingest data in real time, index different data types and run retrieval and ranking algorithms.



A diagram of how to run hybrid search in Rockset. Rockset has indexing, retrieval and ranking built into its vector database.

Rockset for hybrid search: Converged Indexing

With Rockset, you can build vector indexes without impacting live search applications. Its Converged Index provides the flexibility to index any data, including vectors, text, document, geo and time series data, and apply ranking and scoring using SQL. As a result, users of Rockset build and iterate on hybrid search applications faster to drive the most relevant experiences. Rockset is designed for hybrid search at scale:

- Update indexes without impacting live search performance
- Scale out to support multi-tenant applications at high concurrency
- Ranking is as easy as a SQL ORDER BY clause

Architecture of Converged Indexing

The Rockset Converged Index applies elements of a vector index, search index, columnar store and row store on top of a key-value store abstraction. Rockset uses a cost-based query optimizer to exploit multiple indexes in parallel for the most efficient query execution.

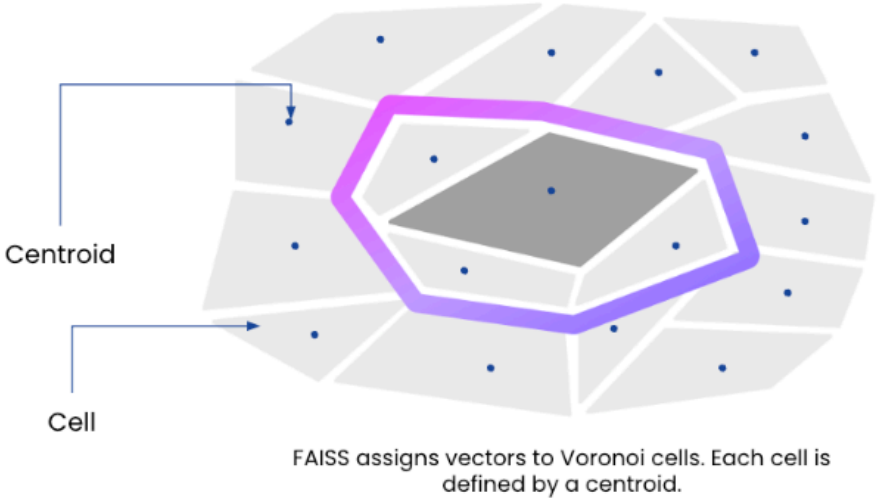
Under the hood, Rockset uses RocksDB as its storage layer. Each document stored in the Converged Index maps to many key-value pairs in the key-value store. This allows for fast writes and field-level mutability, supporting real-time updates in milliseconds.

Vector index

Rockset is designed to be indexing algorithm agnostic and currently supports FAISS-IVF. FAISS is a vector indexing library open-sourced by Meta, and IVF is its inverted file index implementation. Rockset builds a distributed FAISS vector index that supports immediate insertion and recall.

Rockset stores and indexes vectors alongside text, JSON, geo and time series data within the same collection. Users can create a vector index on any vector field(s). FAISS-IVF uses cell-probe indexing to partition the vector space into Voronoi cells,

each represented by a centroid, which is the central point of the partition. The centroids are computed using a sample of the dataset. Vectors are then assigned to a partition or cell based on their proximity to a centroid.



A depiction of the Voronoi cells in FAISS. The centroid is the center of the cell. At search time, cells closest to the search query vector are searched to generate the results.

When indexing vectors, Rockset makes use of the columnar index part of its Converged Index to scan all of the vectors, reducing the initial index build time. As new records are added, they are immediately assigned to the cell based on the vector index data distribution. The vector index also adds fields to each record to store the closest centroid and the residual, the offset or distance from the closest centroid, for search.

Number	Centroid Vector
1	Ctd 101
2	Ctd 102
3	Ctd 103
...	....
1000	Ctd 109

Create FAISS-IVF index and store centroid identifiers in memory

Centroid	Residual	Name	Age	Location	Vector Embedding
1	1010010111	Edwin Jarvis	49	Malibu, California	[.....]
1	1111000111	Samantha Morton	46	Los Angeles, California	[.....]
3	11000100	Marvin Adams	23	Everywheresville, United Kingdom	[.....]

With reach record, compute and store the centroid and residual

Rockset creates a posting list of the centroid vector which is stored in memory. Each record in the collection also contains hidden fields of the centroid and the residual.

FAISS-IVF uses a search index as part of its implementation and there are benefits to this approach: it is simpler, it scales better and it can store metadata efficiently. FAISS-IVF has a shorter build time compared to graph algorithms that require layers of building. While FAISS-IVF query runtime is middle of the pack against other algorithmic approaches, it is well designed for reading data from disk. In contrast, graph-based algorithms are fast in memory but traversal begins to fail as the index is read from disk.

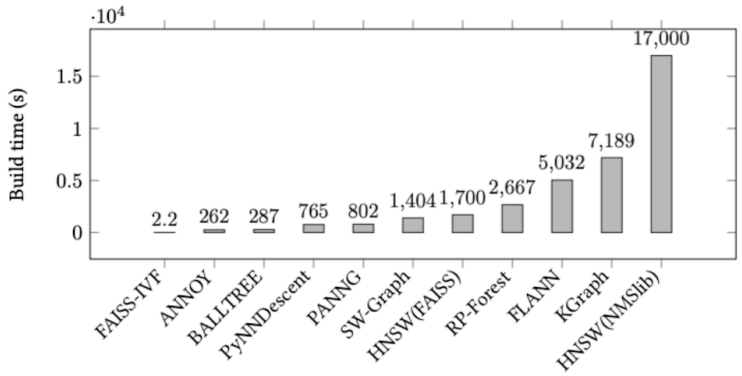
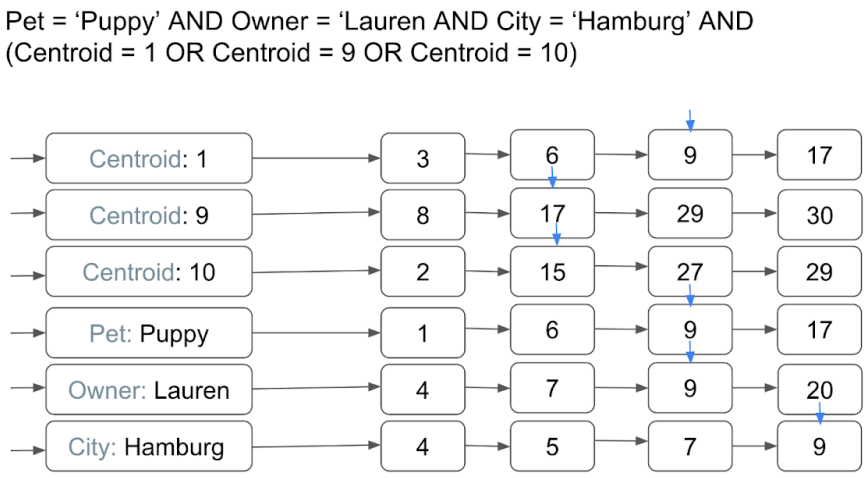


Figure 10 Index build time in seconds for dataset GLOVE. The plot shows the minimum build time for an index that achieved recall of at least 0.9 for 10-NN.

Source: [ANN benchmarks]

FAISS-IVF fits well into the existing search index structure of [Rockset's Converged Index], making it ideal for hybrid search applications. Rockset treats centroids as "terms" and maps values within the centroid to their respective documents. The image below shows the concept of posting lists and how it can be used to execute both term and vector searches efficiently.



Posting lists containing each search term or centroid. In this example, segment 9 would be searched as it contains the Centroid:1 and the keyword terms.

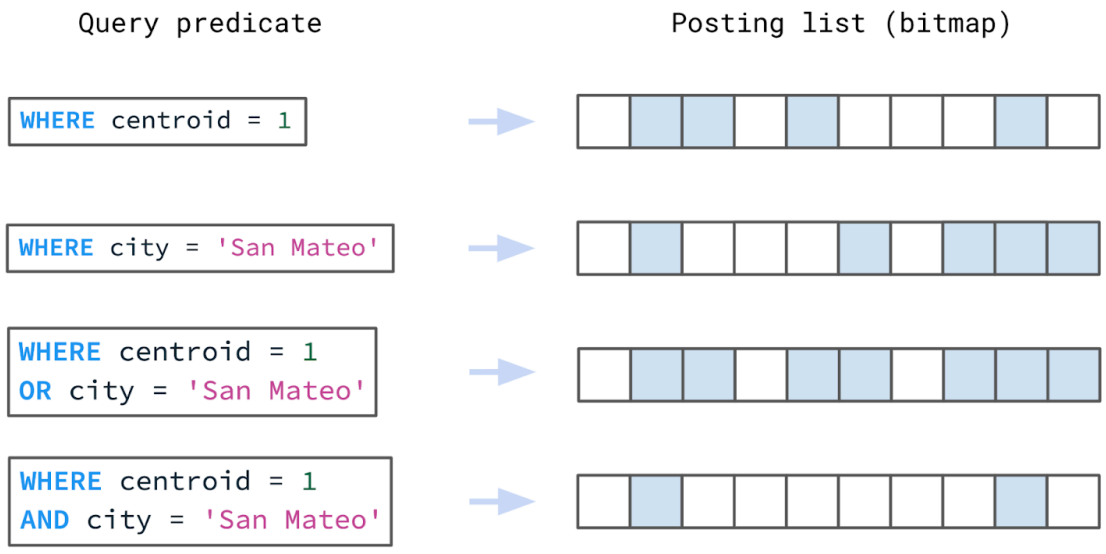
Rockset shards its vector index, training a vector index per-shard. At query time, top K results are retrieved from each shard and then aggregated to produce the search results. Sharding is optimized for latency over throughput, making it ideal for real-time systems at scale.

Rockset allows users to balance recall and speed for their AI applications. During similarity index creation, the user can determine the number of centroids, with more centroids leading to faster searches but increased indexing time. Users can specify quantization values, compression values and vector dimensions. During querying, the user can also choose the number of probes and the number of cells to search, balancing speed and search accuracy.

Search index

Posting lists are fundamental to search database design, and Rockset’s search index uses roaring bitmaps, compressed bitmaps that are optimized for both space and

efficiency, to store posting lists.



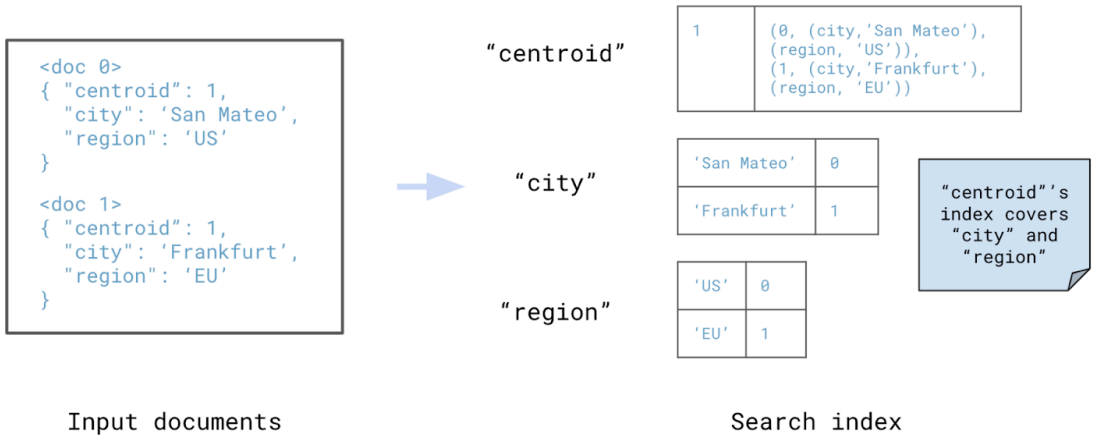
A posting list using roaring bitmaps. This example shows the segments that would be searched intersecting and unionizing the posting lists.

Roaring bitmaps are optimized for performing set operations. For example, `SELECT * FROM foo WHERE centroid = 1 AND city = 'San Mateo'` will intersect the posting lists corresponding to `(centroid, 1)` and `(city, 'San Mateo')`. This example demonstrates that Rockset is ideal for hybrid search, where different data types need to be intersected at query time.

Iterating through a posting list involves iterating one bitmap at a time, each of which covers a group of document IDs. By processing groups of document IDs across multiple posting lists in parallel, Rockset strikes a sweet spot between document at a time (DAAT) and term at a time (TAAT) processing. Traditionally, DAAT involves iterating one document at a time across multiple posting lists in parallel. TAAT involves reading one full posting list, processing it, passing it as a filter to the next one, and then repeating until all posting lists have been handled. Rockset smooths the gap between the two approaches by processing groups of document IDs at a time.

Rockset utilizes covering indexes as part of its search index design to accelerate vector search performance. Covering indexes offer the ability to store hit data in the search index. As a result, all data required for a query can be fetched solely from the search index, without needing to access row or column store data. Rockset stores residuals in the covering index of the centroid entries in the search index so vector search is handled with just one index lookup.

With covering indexes, posting lists now contain document IDs and `(field, value)` pairs for covered fields. Going back to the previous example, searching for `(centroid, 1)` will now return `[(doc ID 0, (city, 'San Mateo'), (region, 'US')), (doc ID 1, (city, 'Frankfurt'), (region, 'EU')), ...]`.



A covering Index for the "centroid" contains additional information, including the "city" and "region", avoiding a lookup step for the added fields during query execution.

Rockset's search index is optimized for join queries, making it easy to combine multiple collections or datasets together. For hash joins and nested loop joins, if the data access for either side of the join is selective, the search index will be used to fetch the rows for that side of the join before applying the join operation. In addition, lookup join is a join strategy that leverages the search index in a novel way. For example, consider the join `A INNER JOIN B ON A.x = B.x WHERE B.y = 5`. If the

predicate `B.y = 5` is highly selective and causes the right side of the join to contain only a few (< 100) rows, Rockset uses a lookup join to transfer the ~100 values of `B.x` to the compute nodes serving the shards of collection A. For each of those values, Rockset uses the search index to perform an efficient lookup for whether a matching `A.x` exists, and if so, emit a match for further processing or returning of the results.

Rockset also joins on vector similarity across collections. This involves splitting input from one side into chunks and joining these input vectors against a collection of documents using a batched similarity search. This type of join is useful for isolating documents associated with a concept across collections and can easily be used to support multi-modal models where metadata can be split by modality.

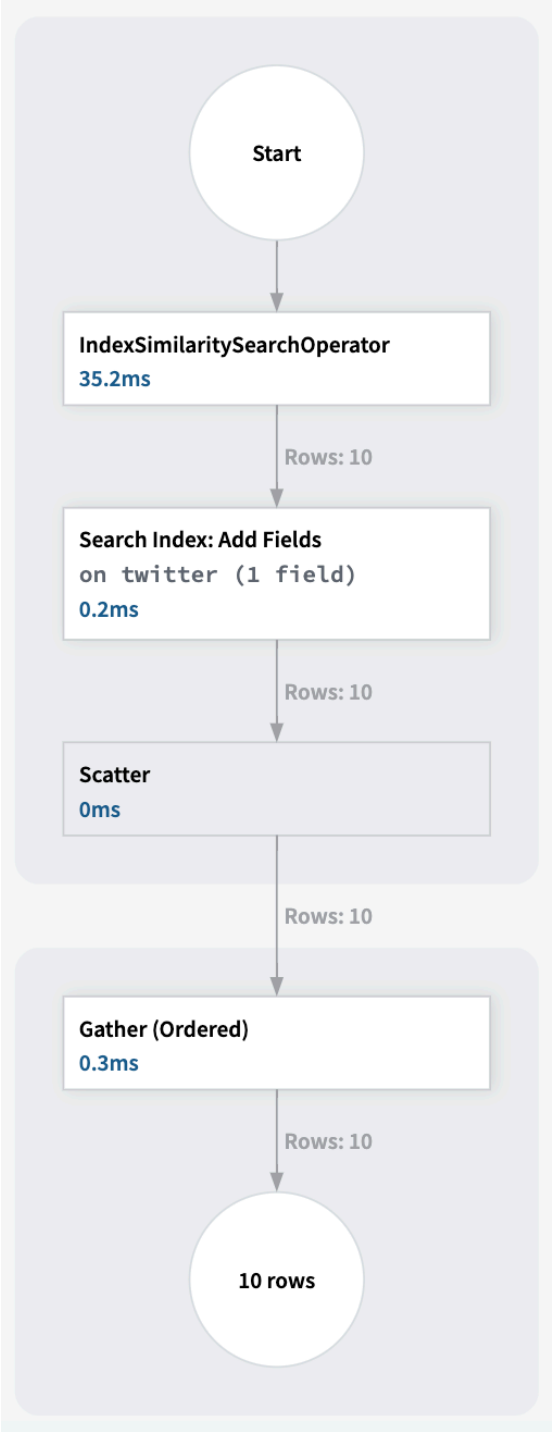
Vector Search

To fully reap the benefits of the Converged Index, Rockset's cost-based query optimizer analyzes the query and the underlying data distribution statistics to identify the most efficient data access path.

Fundamentally, there are two data access paths supported in Rockset for vector search with metadata filtering:

- Pre-filter: Apply filters and then run an exact nearest neighbor search to return the K nearest neighbors to a query vector.
- Single-stage filter: Run an approximate nearest neighbor search and iterate through the centroids until K nearest neighbors are returned.

For a given search, Rockset's cost-based optimizer explores the query plans and chooses the plan with the lowest cost. Choosing the optimal data access path predominantly depends on estimating the selectivity of the data access. For example, a vector search query "Give me 5 nearest neighbors where <filter>?" would need to weigh the different filters and their selectivity, then reorder, plan and optimize the search.



The query profiler in Rockset leveraging both the vector index and search index for execution.

A highly selective query is a query that fetches a small percentage of rows from the underlying collection. In this scenario, it would be more cost-efficient to pre-filter and apply the filters first before running an exact nearest neighbor search. In this situation, Rockset automatically leverages only the search index to return the results.

If the filter is not highly selective, Rockset will use single-stage filtering. It will iterate through the closest centroids to the query until it finds the K nearest neighbors. In this scenario, Rockset's query optimizer asks FAISS for the closest centroids to the query embedding using the specified number of probes for guidance. Probes determine the upper limit of centroids considered during the search process. Under the hood, Rockset applies the centroids as additional query filters to the WHERE clause.

```
SELECT
    city,
    Region,
    APPROX_DOT_PRODUCT(:search_query,
tweet_embedding)
    as similarity
WHERE
    location = "San Mateo, California"
FROM
    example
ORDER BY
    similarity DESC
LIMIT
    10
```

```
SELECT
    city,
    Region,
    APPROX_DOT_PRODUCT(:search_query,
tweet_embedding)
    as similarity
WHERE
    location = "San Mateo, California" AND
    posting_list in [2, 4, 1000]
FROM
    example
ORDER BY
    Index distance (index
    transform(:query), _vector_code)
LIMIT
    10
```

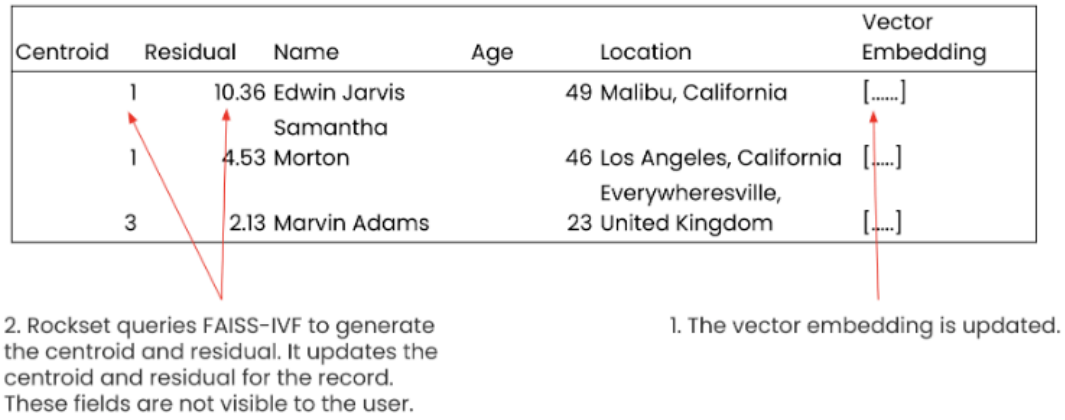
An example of how Rockset rewrites a vector search query. Rockset's query optimizer asks FAISS for the closest centroids to the query embedding. It orders by the distance between the query and the stored vectors.

Single-stage filtering is more computationally efficient than a post-filtering operation where all centroids are scanned before filters are applied. Rockset does not support post-filtering operations for this reason.

Rockset collects statistics on the data distribution for a collection which are maintained continuously, and these are used to estimate the selectivity of data accesses needed in a hybrid search query.

Real-time Updates

Rockset supports real-time updates to vectors and metadata. Rockset is mutable at an individual field level so an update to the vector on a single record will result in an instruction to FAISS-IVF to generate a new centroid and residual. When this occurs, Rockset only reindexes the centroid and the residual for the updated vector field, allowing this operation to occur in less than 200 milliseconds.



Real-time updates in Rockset. When a field value is updated, in this case the vector embedding for Edwin Jarvis, only the individual embedding field is updated rather than the entire document. Rockset instructs FAISS-IVF to generate a new centroid and the residual. These are hidden fields that are used in vector search.

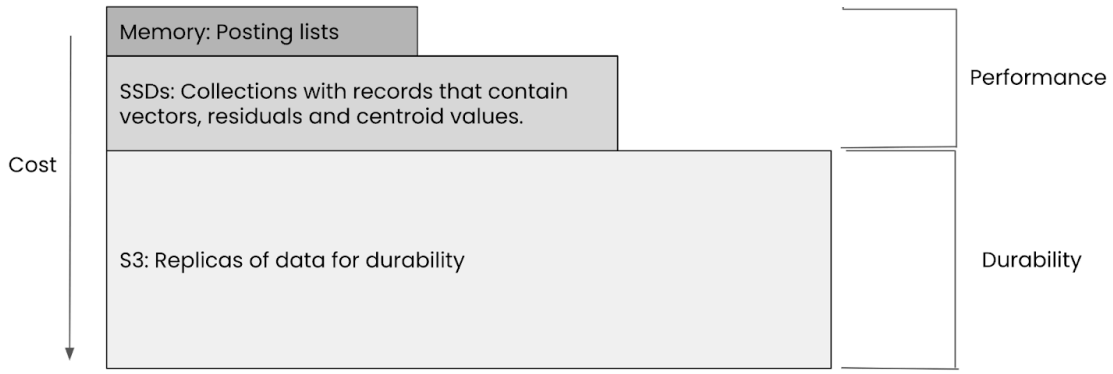
To support real-time updates, Rockset's Converged Index separates the logical search index from its physical representation as a key-value store, which is unique in search database design. Indexes are mutable because each key refers to a

document fragment, meaning that a user can update a single vector or metadata field in the document without triggering a reindexing of the entire document.

Traditional search databases tend to suffer from reindexing storms because even if a single vector or metadata is updated in a large document, the entire document must be reindexed.

Tiered Storage of Indexes

Rockset takes a disk-based approach to indexing. Rockset shards its indexes, applying a local index to each shard stored on SSDs. Once the local index is created, it is uploaded to S3 for durability. For vector search, Rockset only keeps the posting list in memory, greatly reducing the memory footprint for better price performance.

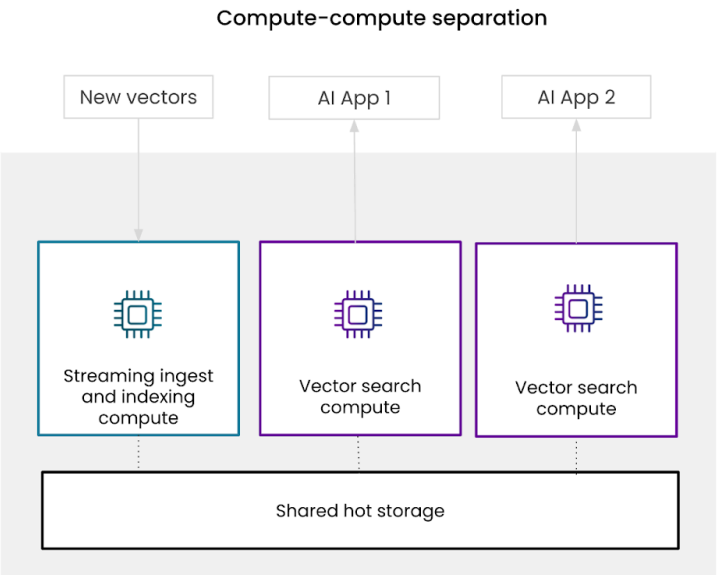


Rockset's tiered storage which makes use of cloud storage for performance and durability.

Isolation of resources for indexing and search

Rockset's compute-compute separation ensures that the continuous streaming and indexing of vectors will not affect search performance. In Rockset's architecture, a virtual instance represents a cluster of compute nodes. Virtual instances can be

used to either index data and/or handle query workloads. Multiple virtual instances can simultaneously access the same dataset, eliminating the need for multiple replicas of data.



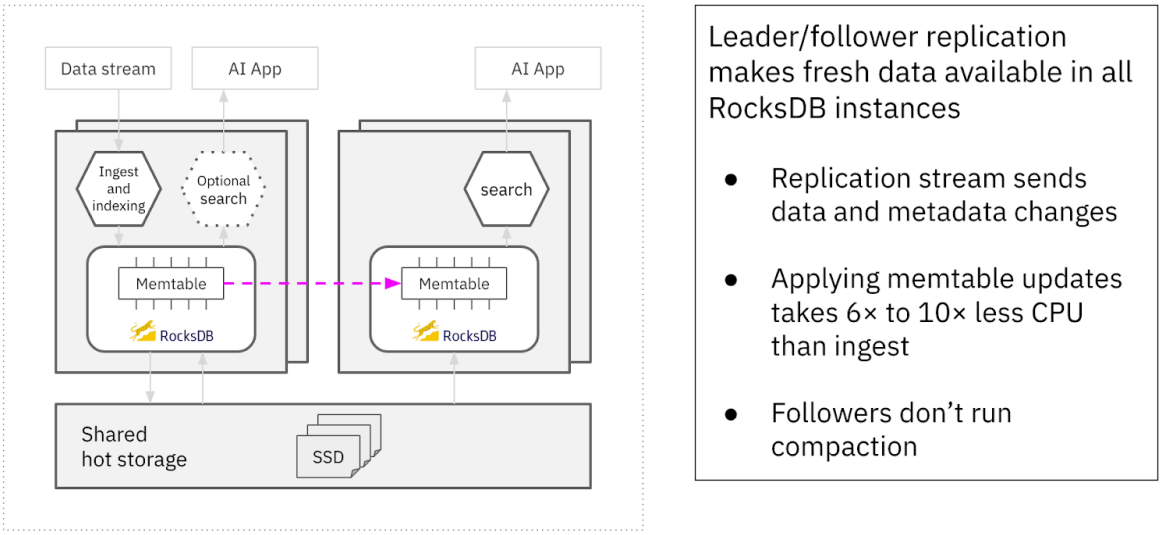
A high level overview of compute-compute separation. Shows that virtual instances can be used to isolate ingestion and indexing from query processing.

Rockset is designed so virtual instances that perform ingestion tasks are completely isolated from those that perform query processing. Thus, the data ingestion, transformation and indexing code paths work independently from the query parsing, optimization and execution in Rockset.

For data to be shared across multiple compute units in real time, Rockset uses RocksDB. In addition to being a popular key-value storage engine, it's also a popular Log Structured Merge (LSM) tree storage engine. In LSM Tree architectures, new writes are written to an in-memory memtable. These memtables are flushed, when they fill up, into immutable sorted strings table (SST) files.

Rockset designed compute-compute separation to be real time by replicating the in-memory state of the memtable in the RocksDB "leader" performing the ingestion, indexing and compaction into the memtables in the RocksDB "follower" instances

that serve queries. This makes fresh data available in single-digit milliseconds across all RocksDB instances that are following the “leader” instance. This implementation means that the compute-intensive ingestion work of indexing and compaction happens only on the leader, avoiding redundant compute expense in the followers.



A detailed diagram of compute-compute separation. Updates to the “leader” memtable are immediately made available to “follower” memtables, enabling follower virtual instances to access the latest data. The architecture of compute-compute separation designates the leader to run ingestion, indexing and compaction.

Compute-compute separation makes it possible for Rockset to support concurrent indexing and search. Compute-compute separation also ensures that users can keep their vector indexes’ recall high by retraining them on the ingester instance when needed without interfering with search workloads.

It’s well known that periodically retraining the index can be computationally expensive. In many systems, the reindexing and search operations happen on the same cluster. This introduces the potential for indexing to negatively interfere with

the search performance of the application. With compute-compute separation, Rockset avoids this issue for predictable performance at any scale.

Multi-tenant design

Rockset allows users to partition records in an index by tenant. At collection creation time, users can specify tenant partitioning field(s) such as `tenant_id`. Rockset stores the tenant (128 bit hash of all tenant partitioning fields) at the head of the search index key so the search index can use the information to reduce the search space significantly. For vector search, this means that although a vector index is built across all records, the search space at query time is reduced dramatically by filtering on the `tenant_id` field, speeding up search performance.

With Rockset's compute-storage separation, Rockset users can create multiple virtual instances, or isolated compute and memory resources, per tenant or for multiple tenants to ensure predictable query performance at scale.

Additional Indexes

Search Index Design for BM25

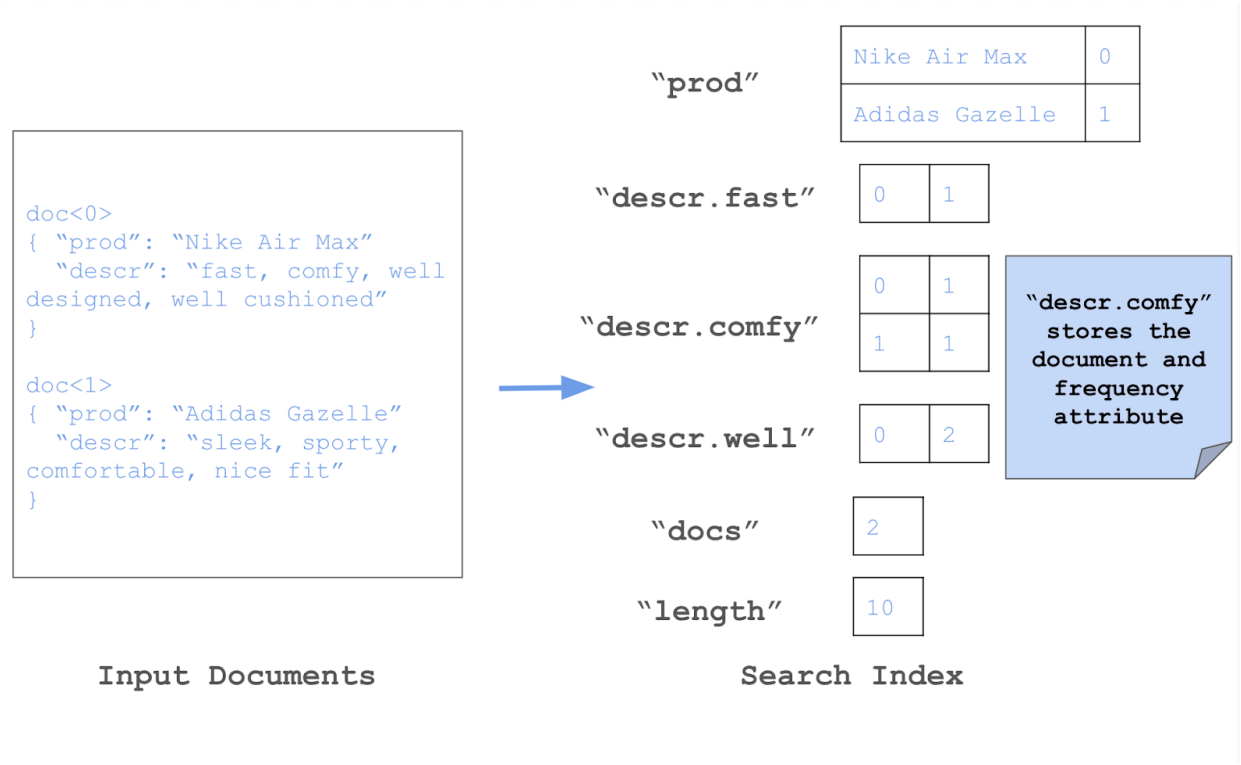
BM25 is a ranking function used to estimate the relevance of documents given terms from a search query. BM25 leverages a bag-of-words approach by ranking documents based on the search terms appearing in each document, regardless of term proximity.

The Rockset Search Index allows for the storage of well-known attributes within the index itself, computed during the indexing process. In the case of BM25, Rockset computes and stores the term frequency per document as a well-known attribute, so for each term-document pair in the search index, Rockset tracks the frequency of the term within the document.

Rockset also tracks two values at the collection level: the total number of documents and the running sum of the total document length, which allows us to easily compute the average document length. Given the use of well-known attributes and

collection-level metadata, Rockset can calculate BM25 scores for individual documents with minimal computational overhead.

```
Unset
BM25(
  terms,
  field
) [OPTION(k=1.6)] [OPTION(b=0.75]
```



Rockset stores the term frequency per document in the search index as a term-document pair.

Search Index Design for Geo Search

Rockset's search index also supports indexing geography values. Typical geospatial queries are not usually searching for exactly one point, but for some compact region of points, like all points within a given distance, or within a polygon. To serve this need, Rockset repurposed the search index to work differently for geographies. First, Rocksert partitions the surface of the earth into a hierarchical grid of roughly square

cells using the S2 library. For each point in a collection, Rockset adds an entry in the search index for each cell which contains it. Since these cells form a hierarchy, a single point is contained by many cells- its immediate parent, and all of that cell's ancestors. This increases space usage, but pays off with better query performance.

Columnar Store for Analytics

The column index stores all values for a particular column contiguously on storage. A query can efficiently fetch exactly the columns that it needs, which makes it ideal for analytical queries over wide datasets. Additionally, column-oriented storage has better compression ratios. Values within one column are usually similar to each other, and similar values compress really well when stored together. There are some advanced techniques that make compression even better, like dictionary compression or run-length encoding.

Row Store for Lookups

The row index refers to storing data in row orientation, which is fairly standard in databases. It optimizes for row lookups and is how Postgres and MySQL are organized.

Ranking Design

Ranking with Reciprocal Rank Fusion

Reciprocal Rank Fusion (RRF) provides an effective method for combining document rankings from multiple search modalities like vector search, text search, and geospatial search. RRF sorts documents according to a proven scoring formula and reduces the need to normalize scores across different search modalities.

The formula balances contributions from various search modalities, allowing for a more nuanced and comprehensive ranking of documents across different search engines. Rockset achieves this with a new SQL function:

```
Unset
RANK_FUSION(
  score [DESC|ASC] [WEIGHT weight],
```

```
  ...
) [OPTION(k=60)]
```

Rockset's RANK_FUSION function executes in memory during the last stage of the query execution process.

Ranking with Linear Combination

Linear combinations provide a ranking mechanism by summing outputs from different search modalities with constant coefficient weightings.

```
Unset
(:alpha * score1) + ((1-:alpha) * score2)
```

By adjusting the coefficient, users can fine tune the influence of each modality, enabling a flexible and customizable ranking system. Linear combinations are typically used to combine scores across search modalities with normalized scores as the output.

Hybrid Search Queries

Vector search and text search

Vector search finds similar items but can miss relevant keywords, that's why many applications use a hybrid approach, combining vector search and text search, to improve the relevancy of results.

Rockset supports hybrid vector and text search. **`BM25`** scores documents based on the frequency and distribution of query terms, providing a measure of text relevance. Both **`BM25`** and **`APPROX_DOT_PRODUCT`** return normalized scores, allowing users to combine the outputs using a linear combination to create a hybrid score.

```
Unset
SELECT
    tweet,
    :alpha * APPROX_DOT_PRODUCT(:search_embedding, tweet_embedding)
         + (1 - :alpha) * BM25(:search_terms, tweets_tokens)
         as hybrid_score
FROM
    twitter t
ORDER BY
    hybrid_score DESC
LIMIT
    10
```

*A SQL example that combines the scores from vector search and text search using an **`alpha`** parameter to weigh the contributions.*

`EXPLAIN` plan:

```
Unset
select tweet:$23, hybrid_score:$25, text_score:$18
  sort $50 desc limit 10
    project $25=(add_(multiply_(0.7, $18), multiply_(0.3, $22)))
      hash inner join on ($21 = $19)
        reshuffle on_aggregators(1)
          add fields on commons.twitter: fields($21=_id, $23=tweet)
            index similarity search on commons.twitter: fields(),
                  $22=kInnerProduct(simidx:rrn:simidx:usw2a1:cc937023-8b
                     4f-4879-a279-7c24e00c1222, [1.50781, 0.230469,
            0.0649414, 0.0397949, -0.0228271, -0.291016, -0.458984,
            -2.07424e-05, -0.196289, 0.296875, -0.503906, ...,
            0.0515137, -0.15332, -0.275391]), query(all)

        reshuffle on_aggregators(1)
          $18=bm25 operator with avgdl on $11 for query
         ['college','basketball'] on tweet_tokenized, $19=_id
            project $11=divide_($10, $5)
                  aggregate sink on (): $10=sum($9) hash ($8) grouping
         ($7)
```

```
      aggregate source on (): $9=sum($6) hash ($8) grouping
($7)
               document length lookup $6 ON
commons.twitter:tweet_tokenized
     aggregate sink on (): $5=sum($4) hash ($3) grouping ($2)
       aggregate source on (): $4=sum($1) hash ($3) grouping
($2)
                 document count lookup $1 ON
commons.twitter:tweet_tokenized
```

`BM25` calculates the relevance score based on text search. `APPROX_DOT_PRODUCT` computes the similarity score based on vector search. The `hybrid_score` combines these scores with an `alpha` parameter to balance their contributions.

By adjusting the **`alpha`** parameter, users can control the influence of text relevance and vector similarity in the search results. The hybrid approach enhances the search by leveraging the strengths of both text and vector search methodologies.

Vector search and metadata filtering

Rockset supports both exact nearest neighbor (KNN) and approximate nearest neighbor (ANN) searches. The distance between embeddings can be calculated using built-in distance functions **`EUCLIDEAN_DIST, DOT_PRODUCT, COSINE_SIM`**. To orchestrate this in SQL, Rockset employs the **`ORDER BY similarity DESC`** clause to sort results by similarity metrics and the **`LIMIT k`** clause to restrict the output to the top k results. Metadata filtering is integrated using the **`WHERE`** clause to impose specific constraints.

```
Unset
SELECT
    tweet,
    APPROX_DOT_PRODUCT(:search_embedding, tweet_embedding) as similarity
FROM
    twitter t
WHERE
```

```
    t.user.friends_count > 1000
    AND t.user.verified_type = 'blue'
    AND t.place.country_code = 'US'
ORDER BY
    similarity DESC
LIMIT
    10
```

Query of vector search with metadata filtering. The query identifies tweets that align with defined user profiles and geographical locations.

In this example, the query targets tweets from verified users in the United States with over 1000 friends, highlighting popular profiles. The `APPROX_DOT_PRODUCT` function computes the similarity between tweet embeddings in the collection and an embedded search query.

Rockset's cost-based optimizer selects the most efficient approach, choosing between pre-filtering and single-stage filtering. An **`EXPLAIN`** plan for the query details the chosen execution sequence, demonstrating the applied filtering strategy:

```
Unset
select tweet:$5, similarity:$4
  reshuffle ordered by $4 desc on_aggregators(1) limit 10
    add fields on commons.twitter: fields($5=tweet) [estimated rows:
        160]
      index similarity search on commons.twitter:
         fields($1=place.country_code, $2=user.friends_count,
         $3=user.verified_type),
         $4=kInnerProduct(simidx:rrn:simidx:usw2a1:cc937023-8b4f-4879-a2
         79-7c24e00c1222, [1.50781, 0.230469, 0.0649414, 0.0397949,
         -0.0228271, -0.291016, -0.458984, -2.07424e-05, -0.196289,
         0.296875, -0.503906, ..., 0.0515137, -0.15332, -0.275391]),
         query(and($1:string['US'], $2:float(1000.0,inf], int(1000,max],
         $3:string['blue'])) limit 10 [estimated rows: 160]
```

A portion of the EXPLAIN plan for the vector search and metadata filtering query.

The similarity search index is selected on four fields: **`place.country_code`**, **`user.friends_count`**, **`user.verified_type`**, and **`kInnerProduct`**. The fields for metadata filtering, **`place.country_code`**, **`user.friends_count`** and **`user.verified_type`**, are applied before calculating the inner product. Given the selectivity of the filters, the cost-based query optimizer leveraged the pre-filtering strategy.

Vector search and geospatial search

Vector search can be enhanced with geospatial filtering or geospatial ranking. Searches across geospatial data are implemented using Rockset's built-in geographic functions including: **`ST_DISTANCE`**, **`ST_CONTAINS`** and **`ST_INTERSECTS`**.

Geographic functions are integrated into the `WHERE` clause of a SQL query to refine searches based on geographical proximity or containment.

```
Unset
SELECT
    tweet,
    APPROX_DOT_PRODUCT(:search_embedding, tweet_embedding) as similarity
FROM
    twitter t
WHERE
    ST_DISTANCE(
        ST_GEOGPOINT(t.coordinates.lat, t.coordinates.long),
        ST_GEOGPOINT(:search_latitude, :search_longitude)
    ) < :distanceMeters
ORDER BY
    similarity DESC
LIMIT
    10
```

A query with vector search and geospatial search. The query identifies similar tweets within a defined geospatial location. The `ST_DISTANCE` filters tweets within a specified radius from a given point.

`EXPLAIN` plan:

```
Unset
select tweet:$4, similarity:$1
  reshuffle ordered by $1 desc on_aggregators(1) limit 10
    add fields on commons.twitter: fields($4=tweet) [estimated rows: 0]
      limit 10 ordered by $1 desc
        filter on (st_distance(st_geogpoint($2, $3), POINT(-142.433
         -57.6749)) < 16100) [estimated rows: 0]
          add fields on commons.twitter: fields($2=coordinates.lat,
         $3=coordinates.long) [estimated rows: 0]
            index similarity search on commons.twitter: fields(),
         $1=kInnerProduct(simidx:rrn:simidx:usw2a1:cc937023-8b4f-4879-a2
         79-7c24e00c1222, [1.50781, 0.230469, 0.0649414, 0.0397949,
         -0.0228271, -0.291016, -0.458984, -2.07424e-05, -0.196289,
         0.296875, -0.503906, ... 0.0515137, -0.15332, -0.275391]),
         query(all)
```

*The EXPLAIN plan where **`coordinates`** and **`kInnerProduct`** are selected from the **`similarity search`** index*

In this example, a geospatial constraint is applied to filter out records outside a defined range. However, a more nuanced approach involved ranking results by geographic proximity. Reciprocal Rank Fusion (RRF) can be used to combine multiple ranking signals, such as geographic proximity and vector similarity, into a single ranking score.

```
Unset
SELECT
    tweet,
    RANK_FUSION(
        ST_DISTANCE(
            ST_GEOGPOINT(t.coordinates.lat, t.coordinates.long),
            ST_GEOGPOINT(:search_latitude, :search_longitude)
        ),
        APPROX_DOT_PRODUCT(:search_embedding, tweet_embedding)
    ) as hybrid_rank
FROM
    twitter t
ORDER BY
    hybrid_rank DESC
```

```
LIMIT
    10
```

A vector search and geographic proximity query using the RFF ranking algorithm. The hybrid rank considers both the geographic distance and semantic similarity of tweets in sorting the result set.

`EXPLAIN` plan:

```
Unset
select tweet:$2, hybrid_rank:$8
  sort $8 desc limit 10
    project $8=(add_(divide_(1, add_(60.0, $7)), divide_(1, add_(60.0,
         $6))))
      window: $6=rank() order: $1 range between: unbounded preceding, 0
      window: $7=rank() order: $5 range between: unbounded preceding, 0
        reshuffle on_aggregators(1)
          project $5=(st_distance(st_geogpoint($3, $4), POINT(-142.433
          -57.6749)))
            add fields on commons.twitter: fields($2=tweet,
          $3=coordinates.latitude, $4=coordinates.longitude)
              index similarity search on commons.twitter: fields(),
          $1=kInnerProduct(simidx:rrn:simidx:dev-usw2a1:UUID, [1.50781,
          0.230469, 0.0649414, 0.0397949, -0.0228271, -0.291016,
          -0.458984, -2.07424e-05, -0.196289, 0.296875, -0.503906, ...
          0.0515137, -0.15332, -0.275391]), query(all)
```

*An EXPLAIN plan showing that the **`ST_DISTANCE`** calculates the geographic distance and **`APPROX_DOT_PRODUCT`** computes the similarity score. **`RANK_FUSION`** combines these scores into a single ranking score, **`hybrid_rank`**, using the RRF method.*

By combining both geospatial and vector search scores, RRF provides a balanced ranking that takes into account both proximity and semantic similarity, resulting in more relevant search results.

Conclusion

All search is becoming hybrid search. Rockset is designed for hybrid search at scale. With its cloud-native architecture, users can build and retrain vector indexes without impacting live search applications. Rockset has been built from the ground up for indexing with a Converged Index that provides the flexibility to index vectors, text, document, geo and time series data and intersects indexes for the most efficient query execution.

LLMs are rapidly advancing with new skills, abilities and reasoning. Rockset unlocks the value of these powerful models with its unique focus on hybrid search and ability to quickly iterate on indexing, retrieval and ranking. Get started with hybrid search on Rockset today with a free trial and $300 in credits.