SingleStore: Do you need a specialized vector database?
CMU 15-445/645 (Fall 2023)
Specialized Database Systems

- Transaction processing
- Data warehousing
- Time series analysis
- Fulltext search
- ...
- Vector search
Outline

- SingleStore Overview
- Vector Search Overview
- Vector Index at SingleStore
- Vector Search at SingleStore
SingleStore Overview
What is SingleStore?

- SingleStore is a distributed general-purpose SQL database
- HTAP
  - Operational and analytical workloads
  - Can run TPC-H and TPC-DS competitively with data warehouses
  - Can run TPC-C competitively with operational databases
- Cloud-native
- Scale out to efficiently utilize 100s of hosts, 1000s of cores and 10s of TBs of RAM
## Benchmarks

**Table 1: TPC-C results (higher is better, up to the limit of 12.86 tpmC/warehouse)**

<table>
<thead>
<tr>
<th>Product</th>
<th>vCPU</th>
<th>Size (warehouses)</th>
<th>Throughput (tpmC)</th>
<th>Throughput (% of max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDB</td>
<td>32</td>
<td>1000</td>
<td>12,582</td>
<td>97.8%</td>
</tr>
<tr>
<td>S2DB</td>
<td>32</td>
<td>1000</td>
<td>12,556</td>
<td>97.7%</td>
</tr>
<tr>
<td>S2DB</td>
<td>256</td>
<td>10000</td>
<td>121,432</td>
<td>94.4%</td>
</tr>
</tbody>
</table>

**Table 2: Summary of TPC-H (1TB) results**

<table>
<thead>
<tr>
<th>Product</th>
<th>Cluster price per hour</th>
<th>TPC-H geomean (sec)</th>
<th>TPC-H geomean (cents)</th>
<th>TPC-H throughput (QPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2DB</td>
<td>$16.50</td>
<td>8.57 s</td>
<td>3.92 €</td>
<td>0.078</td>
</tr>
<tr>
<td>CDW1</td>
<td>$16.00</td>
<td>10.31 s</td>
<td>4.58 €</td>
<td>0.069</td>
</tr>
<tr>
<td>CDW2</td>
<td>$16.30</td>
<td>10.06 s</td>
<td>4.55 €</td>
<td>0.082</td>
</tr>
<tr>
<td>CDB</td>
<td>$13.92</td>
<td>Did not finish within 24 hours</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Product Overview

- **Data pipeline**
  - **Memory**
    - Fast ingestion
  - **Persistent cache**
    - Fast queries
  - **Cloud object store**
    - Unlimited storage

- **Streaming ingestion**
- **Three-tier architecture**
- **Universal Storage**
- **Multi-model**
- **Shared-nothing**

- **Query aggregator**

- **Users**
  - Skiplist indexing
  - Data nodes
Cluster Architecture

- SingleStore is a horizontally-partitioned, shared-nothing DBMS with an optional shared storage for cold data.
- Aggregators
  - Clients connect to aggregators
  - Handle query optimization and planning
  - Coordinate distributed query
- Leaves
  - Perform most computation
Distributed Query Processing

- Tables are hash partitioned by shard key
- Distributed Join
  - Shard key matching, push down execution to individual partitions
  - Otherwise, redistribute data via broadcast or reshuffle
- Optimizer needs to take into account data movement cost
- Certain queries need to be transformed in order to be efficiently executed
Distributed Query Processing

- SELECT AVG(a) FROM db.t
- SELECT SUM(a), COUNT(a) FROM db_0.t
- SELECT SUM(a), COUNT(a) FROM db_1.t
- SELECT SUM(a), COUNT(a) FROM db_2.t
Hybrid Workloads

- Analytical workloads
  - Scan 100s of millions to trillions of rows in a second
- Transactional workloads
  - Write or update millions of rows per second
- Real-time analytical workloads
  - Running analytics concurrently with high-concurrency point reads and writes
Unified Table Storage For Hybrid Workload

- Efficient for analytical workloads
- Efficient for transactional workloads
  - Operational-Optimized Columnstore
Operational-Optimized Columnstore

- On-disk columnstore LSM + in-memory rowstore segment
- Rows are first written into in-memory rowstore segment
- Flusher flushes a new segment when in-memory rowstore segment is full
- Merger merges segments
- Columnstore segments are immutable
  - DELETE/UPDATE mark rows as deleted in the segment
Optimized For Tiered Storage

- Immutable blobs
- No blob writes are on commit (no files either, only WAL)
- Out-of-order replication
Optimized for Analytical Workloads

- Vectorized execution
- Encoded execution
- Late materialization
Optimized for Operational Workloads

- Seekable encoding
- Segment Elimination
  - In-memory metadata (MIN/MAX/deleted bits/...)
  - Sort key
- Secondary index
- Row-level locking
Full-Query Code Generation

- Queries are parametrized
  - `SELECT a + 1 AS x FROM t WHERE b = "abc"
  - `SELECT a + @ AS x FROM t WHERE b = ^`
- Parameterized queries are compiled to MBC bytecodes
- Interpret MBC while compiling MBC to machine code in the background
- Switch to machine code when compilation completes
Secondary Indexes

- Common indexing approaches for LSM tree
  - External index: extra LSM tree lookup per matched row
  - Per-segment index: O(logn) write amplification

- Index generally have sub-linear search complexity
  - Searching a larger index is cheaper than several small ones

- Index LSM tree
  - Per-segment index + index merger
  - Index merger builds cross-segment indexes on multiple segments
Secondary Hash Index

- Two-level indexes
- Per-segment index
  - Posting lists: value -> [row offset]
- Cross-segment index
  - hash(value) -> [(segment id, posting list offset)]
Figure 3: Two-level secondary index structure. A segment and a global hash table from the corresponding LSM trees are shown here.
Unified Way To Identify A Row Efficiently

- Everything identifies a row by (segment id, row offset)
  - Columnar storage
  - Deleted bits
  - Secondary indexes
    - Hash index
    - Fulltext index
    - Row index
    - Vector index
- Can correlate between them efficiently
Adaptive Table Scan

- Hybrid workloads need to combine different access methods and apply them in the optimal order.
- Static decision made by optimizer doesn’t always work.
  - Cost depends highly on query parameters and encodings used.
Adaptive Table Scan

- Per-partition segment selection
  - segment elimination with index or MIN/MAX
- Per-segment row selection
  - filter reordering in next slide
- Per-block row projection
  - Seek or scan?
  - Use column group?
  - Selective column decoding or send encoded values upstream to AGGREGATE or JOIN
Filter Tree
Filtering

- Different ways to evaluate filters each with different tradeoffs
  - Regular filter
  - Encoded filter
  - Group filter
  - Index filter

- Adaptive filter reordering for each block
  - Each segment estimates the cost of each strategy by timing it on a small number of rows
  - Each block reorders the filters based on the cost estimate and selectivity from previous block
# Vector Search Overview
Vector Search

- Given n vectors and another query vector
- Find k nearest neighbors to the query vector
- Dense vectors in d-dimensional space
- Distance metrics
- Approximate nearest neighbors (ANN)
Representation Learning

- Learn to represent objects with vector embeddings
- Semantic similar objects are closer to each other
Retrieval Augmented Generation (RAG)

- LLMs are inefficient and costly to train/fine-tune
- RAG as a cost-efficient approach to GenAI
  - Up-to-date knowledge
  - Domain-specific knowledge
  - Source citation
Vector Search vs Fulltext Search

- Fulltext search relies on keyword matching and can’t capture semantics
  - I like apple
  - I don’t like apple
  - I don’t dislike the fruit company
- Vector search can be multimodal: text, image, audio, video etc
- Vector search is more computationally costly
Vector Index Algorithms

- Tree-Based: KD-Tree
- Hash-Based
- Quantization-Based: IVF, SPANN
- Graph-Based: HNSW, DiskANN, CAGRA
Inverted File (IVF)

- Partition vectors into clusters
- Use the centroids to represent each cluster
Inverted File (IVF)

- Build an inverted index from clusters to vectors
Inverted File (IVF): Search

- Find nearby centroids to the query vector
- Only search within nearby clusters
Hierarchical Navigable Small World (HNSW)

- Skiplist over proximity graph
- Each node is only connected to a small number of neighbors
- Greedy search starts from coarsest layer and refine with finer layers
IVF vs HNSW

- HNSW has higher recall
- HNSW is faster to search
  - $O(\log n)$ vs $O(\sqrt{n})$
- IVF is faster to build
- IVF has much smaller index size
Product Quantization (PQ)

- Vector compression technique that applies to various algorithms
  - IVF_PQ
  - HNSW_PQ
- Not only saves space, but also speeds up distance computation
- Even faster with PQ Fast Scan
- Compression is lossy so need to refine the results for better recall
Index Composition

- In IVF, searching nearest centroids is yet another ANN
- Can build another vector index on centroids: IVF + HNSW
  - Centroids are much smaller
  - Searching nearest centroids requires very high recall
Vector Search Offerings (08/19/2023)

- **Pinecone**
  - Proprietary composite index

- **milvus** / **zilliz**
  - Flat, Annoy, IVF, HNSW/RHNSW (Flat/PQ), DiskANN

- **Weaviate**
  - Customized HNSW, HNSW (PQ), DiskANN (in progress...)

- **drant**
  - Customized HNSW

- **chroma**
  - HNSW

- **LanceDB**
  - IVF (PQ), DiskANN (in progress...)

- **vespa**
  - HNSW + BM25 hybrid

- **Valid**
  - NGT

- **elasticsearch**
  - Flat (brute force), HNSW

- **redis**
  - Flat (brute force), HNSW

- **pgvector**
  - IVF (Flat), IVF (PQ) in progress...

---

Credit: https://thedataquarry.com/posts/vector-db-4/
Vector Index at SingleStore
Overview

- On-disk columnstore LSM + in-memory rowstore segment
- In-memory rowstore segment is small
  - No vector index, just full scan
- Build vector index for on-disk columnstore LSM
  - Per-segment vector index + vector index merger
Per-Segment Vector Index

- Background flusher/merger create a new vector index for each new segment created
- ALTER TABLE creates a new vector index for each segment
- If too many rows are deleted in a segment, its vector index gets rebuild
Vector Index Merger

- Vector indexes have sub-linear search complexity
  - Searching a larger index is cheaper than several small indexes
- Vector index LSM tree
  - Build cross-segment vector indexes on multiple segments
- Vector index is expensive to build so $O(\log n)$ write amplification due to merge can be significant
  - Merge only cold data
Pluggable Vector Index Algorithms

- We are using vector index algorithm as a black box
- This allows us to plug in any vector index algorithm
- In 8.5, we support many popular in-memory vector index algorithms:
  - IVF_FLAT, IVF_PQ, IVF_PQFS
  - HNSW_FLAT, HNSW_PQ
- Post 8.5, we are planning to support on-disk vector index algorithms
- Vector index can be built in an external service
  - Build vector index on GPU
Auto Vector Index

- It’s hard for average users to pick which vector index algorithm to use and to tune various parameters for the given algorithm
- Note that our vector index is always built on immutable data
- We can make smart decisions for the users
- The user just needs to tell us what the requirements are
  - High-recall
  - Cost-effective
Vector Search at SingleStore
Example 1: ANN

SELECT
t.v <-> vector AS d
FROM t
ORDER BY d
LIMIT k;
ORDER BY ... LIMIT Pushdown

- Agg already pushes down ORDER BY ... LIMIT to leaves
  - Currently Merge TopSort, but we can prob do better
- Leaf pushes down ORDER BY ... LIMIT to table scan as a Top filter
Example 1: ANN

Project [t.v <-> vector AS d]
TopSort limit:k [t.v <-> vector]
ColumnStoreFilter [Top(t.v <-> vector, k) index]
ColumnStoreScan t
Example 1: ANN

- Per-partition segment selection
  - Scan all vector indexes within the partition and select top-k for the entire partition
  - Select segments that contain these top-k rows
- Per-segment row selection
  - Top filter evaluates to true iff the row is selected above
- Per-block row projection
Example 2: Pre-Filtered ANN

```sql
SELECT
t.v <-> vector AS d
FROM t
WHERE <filters>
ORDER BY d
LIMIT k;
```
Pre-Filters

- If `<filters>` are executed after vector index scan
  - There will be less rows after filters
  - We can let vector index scan to output more rows at the beginning, but in practice it’s very hard to predict
- `<filters>` need to be executed before vector index scan
  - Make vector index filter aware of its pre-filters
Example 2: Pre-Filtered ANN

Project \([t.v \leftrightarrow \text{vector AS } d]\)
TopSort limit:k \([t.v \leftrightarrow \text{vector}]\)
ColumnStoreFilter \([\text{Top}(t.v \leftrightarrow \text{vector, } <\text{filters}>, k) \text{ index}]\)
ColumnStoreScan t
Example 2: Pre-Filtered ANN

- Per-partition segment selection
  a. Segment elimination with pre-filters
  b. Scan all vector indexes within the filtered segments and select top-l for the entire partition
  c. Run pre-filters on these top-l rows
    ■ If there are at least k output rows, select top-k.
    ■ If there are less than k output rows
      ● Either retry b with a larger l
      ● Or fall back to not using vector index scan
Top Filter

- \( \text{Top}(\text{expr}, \langle \text{filters} \rangle, k) \) is true iff \( \text{expr} \) of this row ranks within the top-\( k \) among all the rows that pass \( \langle \text{filters} \rangle \)
- Top filter is just a regular leaf node in the filter tree
- Can have many Top filters in the filter tree with different pre-filters
- Filters outside of Top filter are post-filters
- Filter reordering can happen within pre-filter tree and post-filter tree
- Retry happens within Top filter
Example 3: Join

```sql
SELECT
t.v <-> vector AS d
FROM t JOIN s
ON t.id = s.id
WHERE <s.filters>
ORDER BY d
LIMIT k;
```
Example 3: Join

Project \([t.v \leftrightarrow vector \text{ AS } d]\)
TopSort limit:k \([t.v \leftrightarrow vector]\)
HashJoin
|---HashtableProbe \([t.id = s.id]\)
|  HashTableBuild alias s
|  ColumnStoreFilter \([<s.filter>]\)
|  ColumnStoreScan s
ColumnStoreFilter \([\text{Top}(t.v \leftrightarrow vector, t.id = s.id, k) \text{ join index}]\)
ColumnStoreScan t
Example 4: Combining Fulltext and Vector Search

- Each query contains multiple subqueries
- Each subquery has its own type: fulltext or knn
- For a given row
  - Each subquery produces a score
  - The final score is a weighted sum of all individual scores
- The query selects rows with the highest final score
Example 4: Combining Fulltext and Vector Search

- Execute each subquery individually as a filter to select rows that have a positive score for that subquery
- Union all rows selected by each subquery
- Compute the final score for all rows in Step 2 and output the highest ones
Example 4: Combining Fulltext and Vector Search

SELECT
    MATCH(t.s) AGAINST ('pattern') AS score1,
    t.v <-> vector AS score2
FROM t
WHERE <filters>
ORDER BY weight1 * score1 + weight2 * score2
LIMIT k;
Example 4: Combining Fulltext and Vector Search

Project [
    MATCH(t.s) AGAINST ('pattern') AS score1,
    t.v <-> vector AS score2]
TopSort limit:k [weight1 * score1 + weight2 * score2]
ColumnStoreFilter [
    (<filters> AND MATCH(t.a) AGAINST ('pattern') index) OR
    Top(t.v <-> vector, <filters>, k) index]
ColumnStoreScan t
More Examples

- Vector index join
- Cross apply
  - Batched workload, good for GPU
Other Vector Index Filters

- Vector range search
  - \( t.v \leftrightarrow \text{vector} > \text{threshold} \)

- Maximal Marginal Relevance (MMR)
  - Representatives of nearest neighbors
  - New neighbor can’t be too close to previously selected neighbors
Thank You