SingleStore

Innovating with storage formats To push the limits in a hybrid database

Eugene Kogan, Software Architect, SingleStore

CDMS 2023

Intro: QP and physical data in traditional OLTP

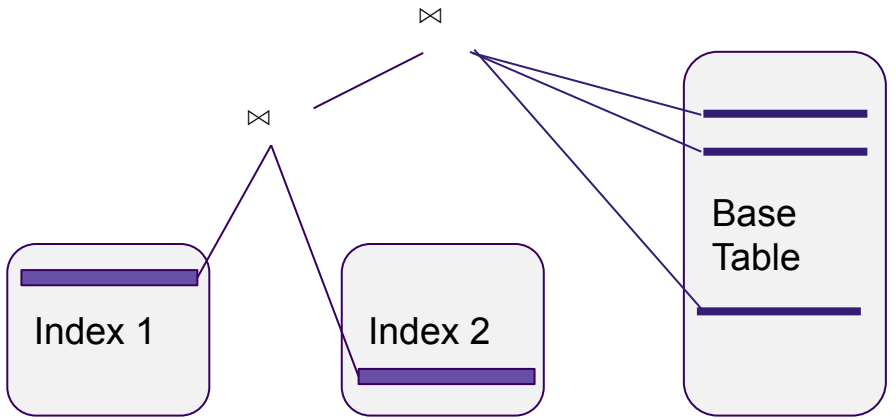
QP may look and cost multiple a physical data representations to

- JOIN: HASH ⇒ MERGE
- AGGREGATION: HASH ⇒ Streaming
- Minimize the read set given Filter/Join predicates

Intro: QP and physical data in traditional OLTP

QP may look and cost multiple a physical data representations to

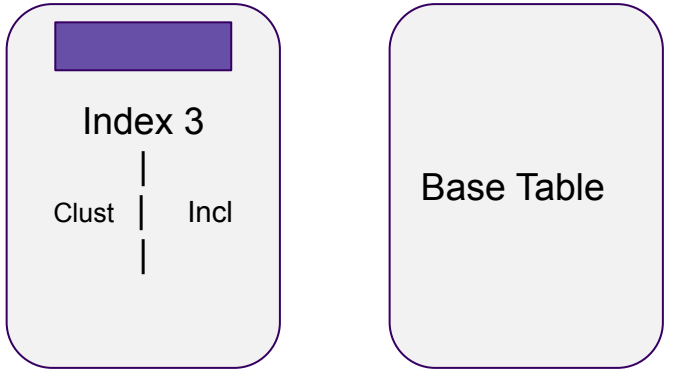
- JOIN: HASH ⇒ MERGE
- AGGREGATION: HASH ⇒ Streaming
- Minimize the read set given Filter/Join predicates



Intro: QP and physical data in traditional OLTP

QP may look and cost multiple a physical data representations to

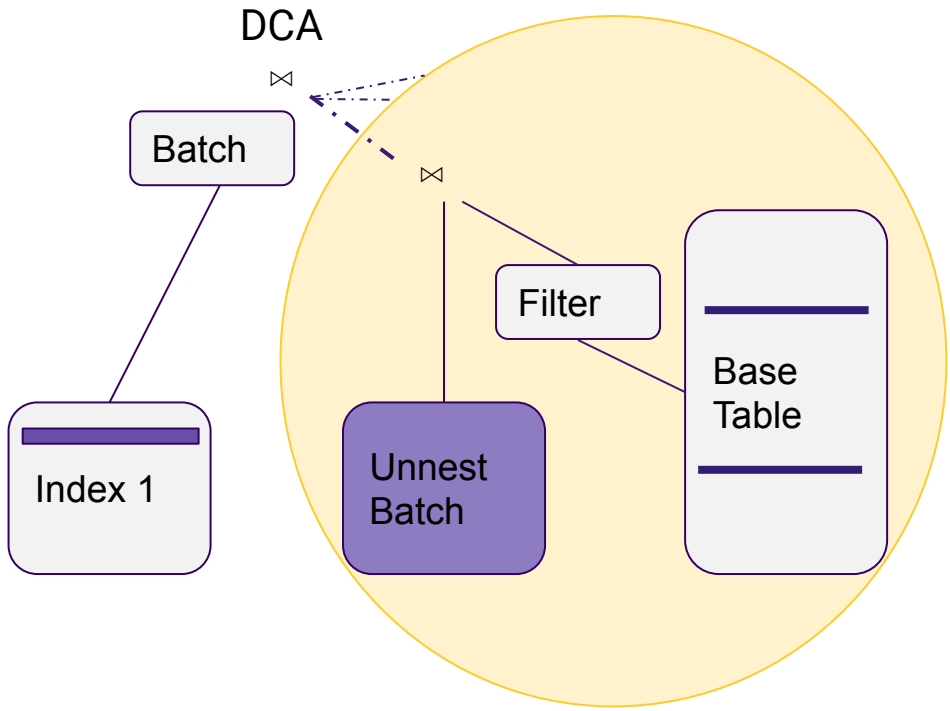
- JOIN: HASH ⇒ MERGE
- AGGREGATION: HASH ⇒ Streaming
- Minimize the read set given Filter/Join predicates



Intro: QP and physical data in New/Distributed OLTP

QP may look for some additional optimizations

- JOIN & AGREGATION: Distributed ⇒ Local
- Minimize the distributed read set given Filter/Join predicates



Intro: QP and physical data in a DW

It's mostly about using metadata (of data in columnar representation) to

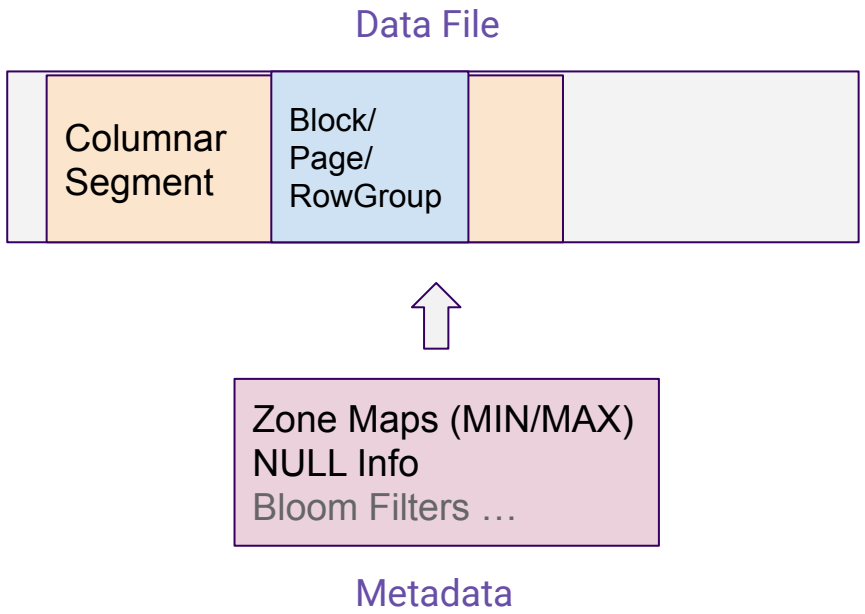
- Simplify & optimize query
- Minimize/prune physical read set on different levels

Metadata contains zone maps (MIN/MAX) and NULL info

- WHERE row.col1 < 3 ⇒ WHERE file.col1_MIN < 3 *

This physical read set pruning may be based on

- Dense metadata stored separate from files
  - Usually happens during query planning
- Metadata encoded together with the columnar data files
  - Normally happens in runtime



If indexes are built they are file/block pruning indexes, not row level (e.g. Search Index) **

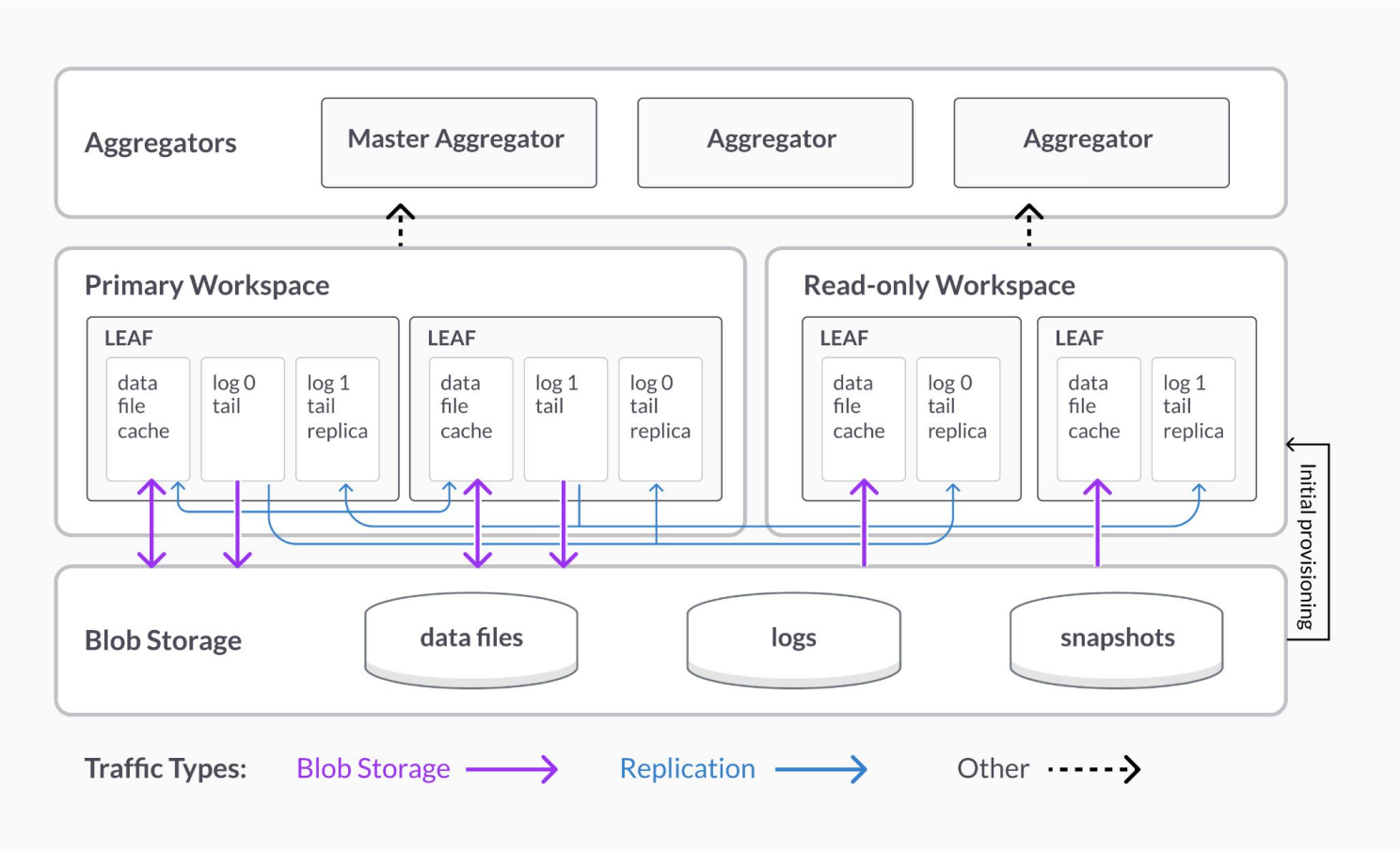
Multiple redundant data copies with different clustering?
Choose one is to maximize pruning efficiency

SingleStoreDB - Transactions & Analytics

- **HTAP, Cloud-native**
  - Operational and analytical workloads
  - Can run all of TPC-H and TPC-DS competitively with cloud data warehouses
  - Can run TPC-C competitively with cloud operational databases
- **Commercially available with over a decade of development**
  - About $100M annual revenue
- **100s of customers with demanding production workloads**
  - Large finance, telecom, energy and tech companies
- **Some properties of the implementation**
  - Support MySQL and Mongo API
  - QP uses both code generation and vectorized processing
  - Supports WebAssembly extensions

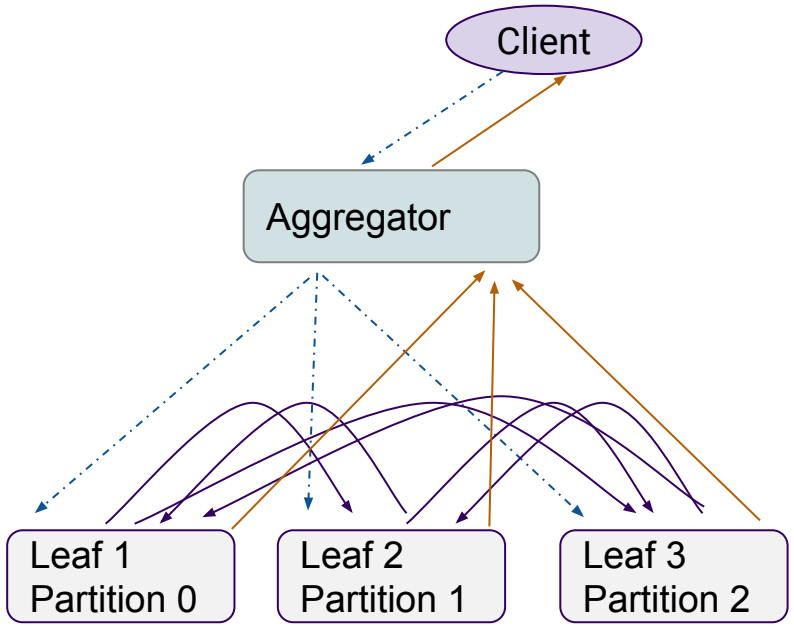
SingleStoreDB - Cluster Architecture & Storage Tiers

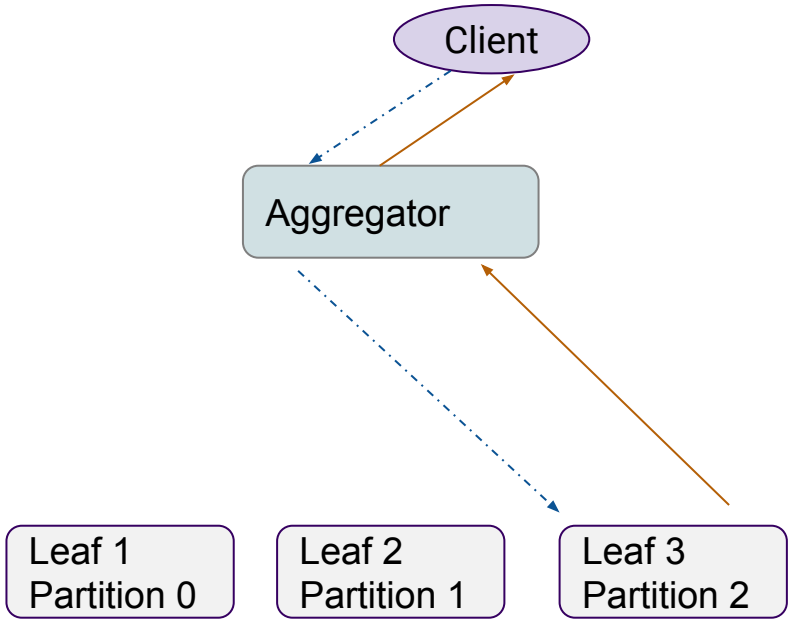
- Nodes in a cluster
  - Organized into workspaces
  - That's where queries are executed
  - Easy to add remove nodes
    To control parallelism/latency
- One writable Primary workspace + zero or more R/O secondary ones
  - Easy to add/remove R/O workspace
- 3-tier storage:
  - RAM (Replicated)
  - Disk (Replicated or cache)
  - Cloud Blob Storage
- No blob writes are on commit latency critical code path
  - No Files either, only WAL



SingleStore

SingleStoreDB - Distributed Query





Transactions & Analytics: what does it really mean?

Queries with different selectivity have competitive latency and cost

- Low selectivity - full scans for large scale analytics
- Medium selectivity - realtime/operational analytics and information retrieval
- High selectivity and point reads - OLTP/KV

Latency should be stable for operational workloads (medium&high selectivity) ⇒ some kind of load isolation

Different kinds of database mutations should have competitive latency and cost

- Bulk insert should be parallel and cheap, with reasonable freshness when querying
- CUD, trickle inserts, events streamed in from realtime sources,
  reasonably sized update and read-modify-write, should be fast (*X*ms) and cheap
- Sparse updates touching a few rows in each partition

Should support read-modify-write, not only event-driven data modelling

- Should support multi-statement transactions

⇒ Fine-grained locking (row locks), MVCC, scans & seeks, indexes, partitioning

SingleStore's approach to LSM Tree

Multi-level

Memory layer - Level 0

- Row-centric data in Lock-free Skiplist
- Multi-versioned, committed & uncommitted
- Also used for locking
- Logged & replicated (with other tables in the partition)

Disk (& Blob Store) resident levels

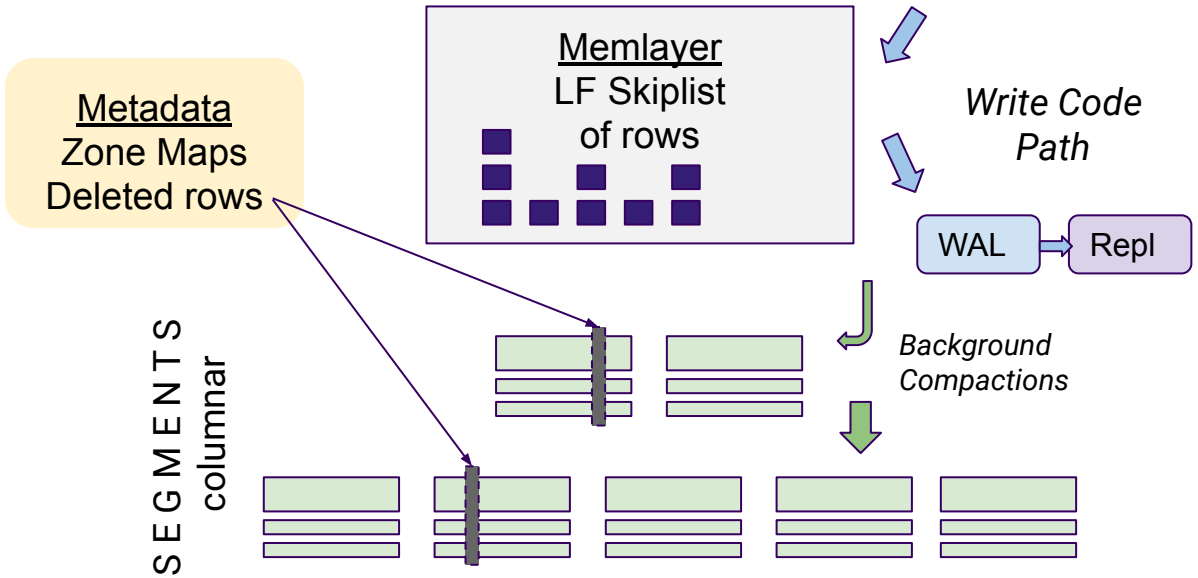
- Column-centric data split into Segments
- Json is shredded using rep-def levels, own encodings

Segment & file Metadata in a system table

- Zone maps (MIN/MAX), NULL info
- Positions of deleted rows in Segments

Balance the cost of deletes and scans in analytical queries

- Tax: find row position in a segment on UPSERT, update Metadata
  - No tax if the row is freshly inserted/updated and is in memlayer
- Win: No CPU-intensive LSMT level merging on unordered scans



SingleStore

Hash Index - compact indexing for equality predicates

Row level index for equality Filter and Join predicates

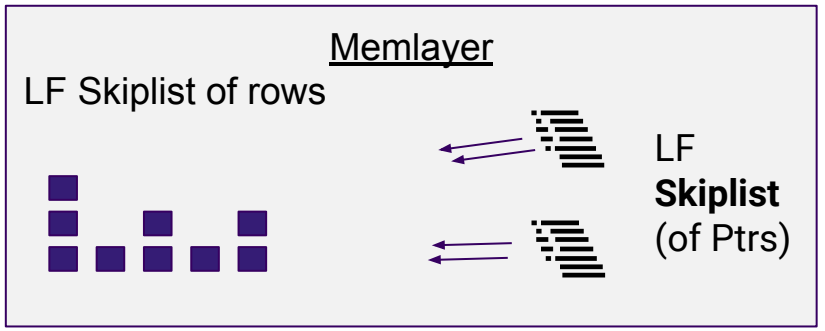
- Zone maps help with range Predicates to some degree
- Use different tech in different memory tiers

Memory layer

- Each index: Lock-free Skiplist of pointers to record versions
  - No value copy for a row to index - just a pointer
- Pick one index when Filters much multiple

Disk (& Blob Store) resident levels

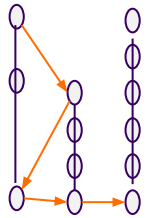
- Dictionary + Posting Lists of row positions
- Immutable and well compressible
  - Single column index size ~= column size
- Complex Filter expr, matching multiple posting lists, efficiently intersected with the likes of skip pointers



Segment S1

Pos	1	2	3	…	99	100
Index column	foo	bar	foo	…	bar	bar
Other column	3	5	8		195	199

Posting List intersection



SingleStore

Hash Index #2

Seekable segment encoding

- To take advantage of inverted indexes
- Divide columnar segment into blocks by # of rows
- Additional metadata
  - Block offset table
  - Block level Zone Maps
- Preference to fixed size column encoding

Global Hash Index

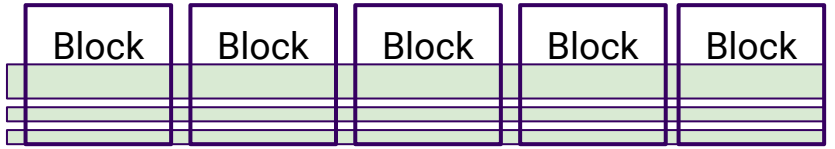
- For Segment selection
- Hash Tables of <Segment_ID>:<posting_list_offset>
  - To minimize # of disk reads
- Organized into a tree similar to the LSM
  - Starts as 1-1 Segment-HT
    And quickly merged into a HT mapping to many Segments

Multi-column Index

- Multiple single-col work surprisingly well, but can build multi-col too

Full Text Index

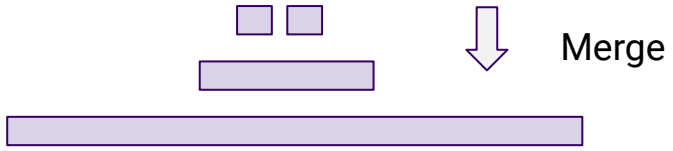
- Will use similar technology in the future
- For now we still use Apache Lucene™ Library



Global Hash Index's HashTable

...	...	...
0x33	S1:17	S2:...
...	...	...
0x59	S1:1	S2:...
...	...	...

Global Hash Index's tree of HashTables



Mix row- and column-centric data with Column Groups

Column Group - row-centic data in a column store

- Stored as a struct/record in an extra column
- Full redundancy for now, if configured
  - Planning a mix of column- and row-centric in the future
  - Design can handle different layout for each segment
- As indexes are positional, they are fully applicable to Column Groups
- 30% improvement on TPC-C (narrow rows), but CRUD on wider rows improved 6-8x

Segment S1

	Offset *1*		*17*	
Inverted index	foo:1,3	…	bar:2,99,100	

	Pos *1*	*2*	*3*	*…*	*99*	*100*
Index column	foo	bar	foo	…	bar	bar
Other column	3	5	8	…	195	199
Column Group (*)	**{foo, 3}**	**{bar, 5}**	**{foo, 8}**	**…**	**{bar, 195}**	**{bar, 199}**

New

Smart Scan: how QP uses physical data structures

Stage 1: Segment Selection (Per Partition)

- Inputs: *Filter predicates, Global Hash Index, Segment Metadata*
  - E.g. combine *LookUpGlobalIndex(idx1, "foo")* and *(segment.col2_MIN < 3)*
- Output: *{SEGMENT_ID [, {value, posting_list_offset}[,...]]} …*

Stage 2: Per Segment's Block Row Selection

- Combine *Posting Lists* (⇐*Local Hash Index*) *and Filters(<unindexed_columns>)*
- Output: *Selection Vector ({2, 3, 7, 9,...} or 011000101…)*
- Feature: Filter predicate reordering, e.g. WHERE idx_col="foo" AND col2<3 AND f(col3,col5)=11
  - Measure actual cost of decoding and function evaluation on Block 1
  - Reorder Block-to-Block based on actual cumulative cardinalities

Stage 3: Row Projection - Per Block (of a Segment)

- Decide on: a) Access Method SEEKs vs SCAN, b) on use of Column Group, c) reconstruct json or just scan a col
- Selective column decoding or send encoded values upstream to Aggregate or Join.

Vector Index for Approx Nearest Neighbor Search

- Based on FAISS
- In development

Overview

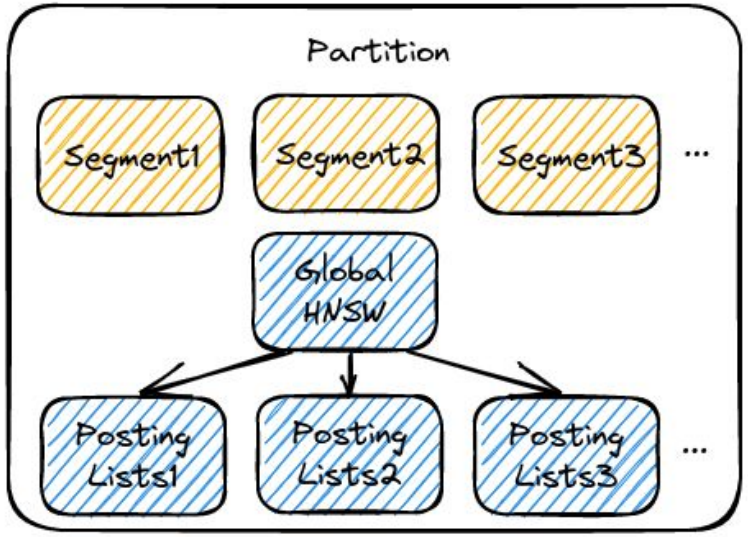
- An HNSW+IVFPQ index is built for each partition
- Vectors are clustered to centroids

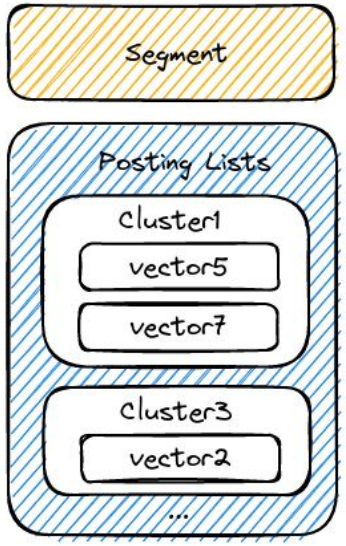
Global Vector Index

- An HNSW index on centroids (in memory)
- Allows for finding closed centroids quickly and accurately

Segment Vector Index

- An inverted index from centroids to list of vectors
- Vectors are PQ-encoded for smaller size and faster distance computation





SingleStore

Smart Scan With Vector Index

Stage 1: Segment Selection (Per Partition)

- Inputs: *Filter predicates, Global Hash Index, Segment Metadata*
  - E.g. combine *LookUpGlobalIndex(idx1, "foo")* and *(segment.col2_MIN < 3)*
- Output: *{SEGMENT_ID [, {value, posting_list_offset}[,...]]} ...*

Stage 2a: Partition-level Top K Row Selection with Vector Index

- Search HNSW: Output <centroid>
- For each Segment: use PQ-compressed vectors to compute global Top K' (K' > K)
  Output {SEGMENT_ID, posting_list}...

Stage 2b: Per Segment's Block Row Selection

- Combine *Posting Lists (⇐Local Hash Index)* and *Filters(<unindexed_columns>)* and ***Stage 2a*** *output*
- Output: *Selection Vector ({2, 3, 7, 9,...} or 011000101…)*

Stage 3: Row Projection - Per Block (of a Segment)

- Decide on: a) Access Method SEEKs vs SCAN, b) on use of Column Group, c) reconstruct json or just scan a col
- Selective row decoding

Projection: traditional index for Distributed SQL

Projection - a Materialized View for transactions/OLTP

- Maintained eagerly - on statement boundary
  - ⇒ Higher update cost for minimal read set
- Projection, Re-sorting, Re-partitioning
  - Selection (== filtered Projections - future)

Projection vs traditional Secondary index with included columns

- Can have different sort order and possibly different Sharding
- Can have its own Hash Indexes
- Can have UNIQUE constraints enforced by indexes
- Can have Column Group independent from the base table
- Can have different table type (not covered here)
- No separate statistics

Use in Query Planning

- Aggregation: Hash ⇒ Streaming
- Join: Distributed ⇒ Local
- Scan: Full ⇒ Range
- Scan: use Hash Index or Column Group

```
CREATE PROJECTION proj1
( SORT KEY (col1, col2),
  SHARD KEY (col1),
  HASH KEY (col3),
  UNIQUE HASH KEY (col1, col11)
)
AS SELECT col1, col2, col3, col11
   FROM base_table;
```

Conclusions

A family of indexing techniques can be fairly cheap to store, maintain and use

- Hash, Vector, Full Text
- Mix row- and column-centric representations
- Just Smart Scan complexity, no impact on Query Planning

Distributed index as a table > traditional secondary index

- This is for OLTP. Different solutions for Analytics and Event Processing.

We have checked every box for HTAP system requirements but more to do

- Indexing nested json elements
- More Vector Indexes
- Materialized Views for analytics and streaming event processing

How should open data format look like to work for an HTAP system?

- Topic for discussion