Choco: Memory Optimized Column Storage Engine

Binglin Chang

Figure 1: chocolate

The tablet data structure looks like a chocolate tablet

Background

Currently, the underlying storage engine in Doris only supports append style batch write, update/upsert/delete by a primary key (like in traditional DBMS/KV) is not supported.

As Doris become more popular, many use-cases start to request the `realtime update` capability, e.g.

- eCommerce use-case: order, inventory status real-time analytics
- Finance use case, use account/balance/transfer checking
- Social media, user/post status updates & statistics

Also, Doris rollup table update on new data batch can be considered as `counter` updates, which may also benefit from this storage engine.

Currently, Doris uses a special `REPLACE` column property to `simulate` upsert semantics. It's basically merge-on-read, when scanning a tablet, Doris automatically merges versions under the same key and only keep the latest version. In real-time use-cases, if the ingestion frequency is high, there are a lot of segments need to be merged, causing a performance bottleneck.

Assumptions & Design goals

We propose a new memory optimized column storage engine to suspport the `realtime` + `frequent update` use-case. We discuss the assumptions and requirements for this storage engine in 3 categories: data ingestion, data query and data storage.

Data Ingestion(Write) side

So write transactions have the following characteristics:

- TXs come as large write-only batches, high throughput is prefered than low latency(<1s), this is typical in OLAP use-cases(CDC, stream data loading, ETL).
- There are only one or a few concurrent load tasks(write TXs).
- In most update use-cases, for all the columns(<50), only a few columns(<5) are updated, e.g. `order` status, `session` update time, counters.

- when perform an update, only the updated columns needs to be specified, not all columns.
- Most updates are on recent `hot` data, `hot` data become `code` after awhile(one to few days), so it's OK to hold recent `hot` data in memory, and later convert to on-disk storage when it become `code`, or just delete them if real-time analytics only query recent `hot` data.
- Like there are `hot` and `cold` rows, there are also `hot` and `code` columns, `cold` columns(never/seldom updated) can reside on disk.
- Deletes are rare, `queue` like use-cases(insert new keys then delete them later) is not considered.

Query(Read) side

- Repeatable snapshot read on recent versions(<1min), for analytical queries.
- In some use-cases, read committed(latest version) is acceptable or preferred.
- Fast analytical scan, so column store(in-memory or SSD) is preferred, write should not block read.
- Schema evolution should be supported, columns(with default value) can be added, non-key columns can be removed online.

Storage planning

Storage Medium

- Data persists on local SSD/NVMe, future design may consider move some in-memory data sturcture into PMEM.
- like the disk based storage engine, data persists on common linux FS, support random read/write.
- No plan to support cloud storages like S3, yet.

Data size assumption

- 100M-1G rows total per day partition(rows, not updates),
- ~10M rows per tablet after hash bucketing
- assuming 100-1000 bytes per row, all the raw data will use 1G-10G memory, plus hash index ~0.1G memory
- so a typical in-memory tablet will consume 1.1G-10G memroy per 10M rows
- In the future, we can flush cold data(old rows and non-updatable columns) onto disk,to keep the memory consumption small.
- For example, asumming a eCommerce Order table have 30M orders per day, a order record may have 50 columns, only 5 columns will be updated(e.g status: ordered, payed, shipped), so only 10% of the data needs to be kept in memory. Further more, most of these orders will never be updated again after the order is finished, those finished orders could also be flushed onto disk, saving more memory.

Goals

Assuming enough memory and CPU, storage on SSD/NVMe

- single-thread load throughput should >100MB/s or 100K rows/s
- scan performance should be equivelent/close-to in-memory array read, something like >2GB/s

Design

User interface

Use `storage_medium` property in `create tablet` statement, user can also specify the number of recent partitons to be stored in-memory, old partitions will be converted to SSD/HDD automatically.

```
create tablet
xxxx
PROPERTIES(
"storage_medium" = "MEMORY",
 );
```

Data structure

Each storage engine instance(MemTablet) conresponds to a tablet replica in Doris. A MemTablet's logical structure looks like this:

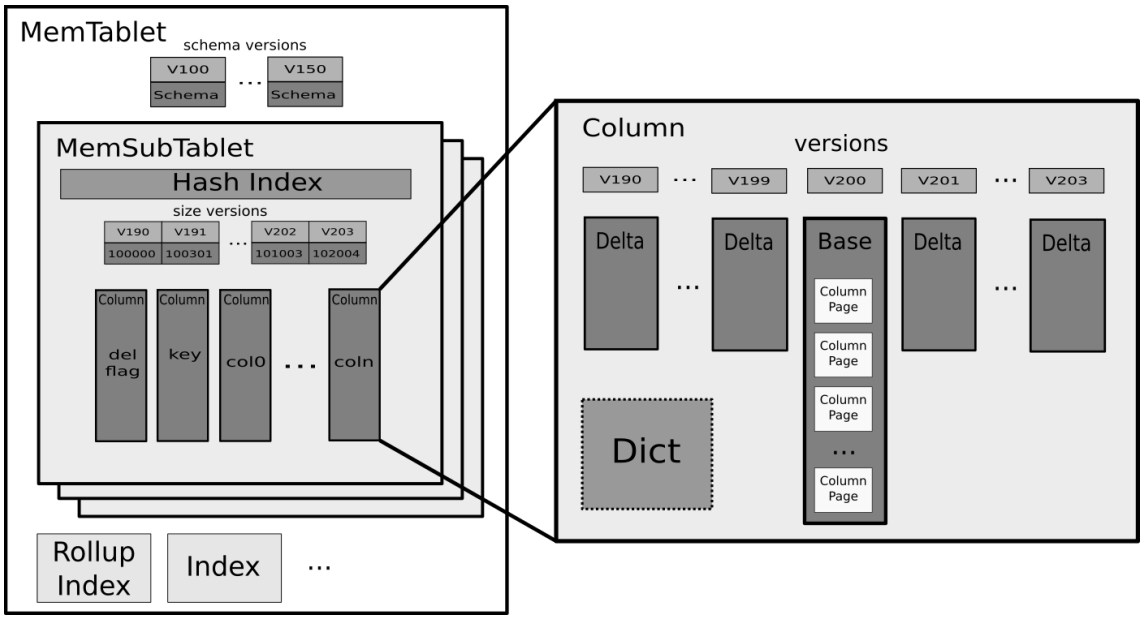

Figure 2: main structure

MemTablet is the root level object, it contains:

- multiple(currently only 1) MemSubTablet, multiple sub-tablet is used to support MDC(Multi-Dimensional Clustering)
- a verison vector to maintain scheme changes
- other indexes(rollup, global dict etc)

MemSubTablet maintain a set of rows, it contains:

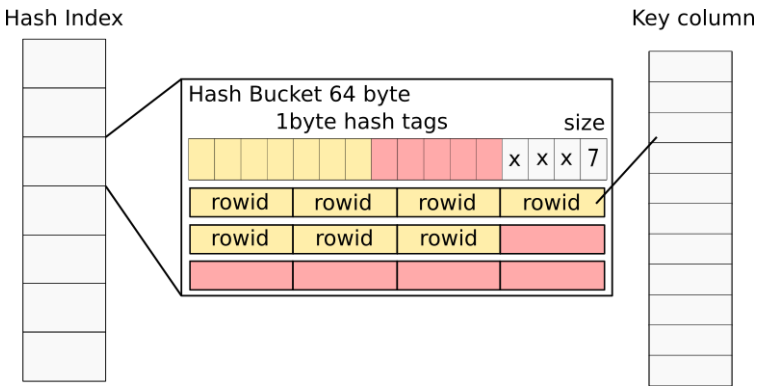

Figure 3: hash index

- a HashIndex, map row key to an uint32 rowid, the desgin is copied from facebook folly F14Hash, to save memory, hash index only store uint32 rowids, not the actual key. Considering a 10M row tablet, with 0.7 fill rate, the hash index will use 10M/0.7/12 = 1.2M buckets * 64B/buckets = 77MB memory.
- a version vector to maintain number of rows(including deleted rows, or size of key space)
- a map of columnid to Column object.
- there is a special Column to mark deleted rows

Column maintains data of a specific column, it contains:

- a verion vector to maintain Base/Delta of each version
- optional Dict if this column is dict encoded

Base store all the data of a column at a specific version

- it divides values into 64K blocks, each stored in a ColumnPage, it can be in-memory or on-disk.

Delta store the changed cells of a column between 2 versions

- it contains an index array to store changed cell's rowid
- and a value array to store changed value
- it also divides its values into blocks coresponding to base's 64K blocks, so index array only need to store 16bit offset

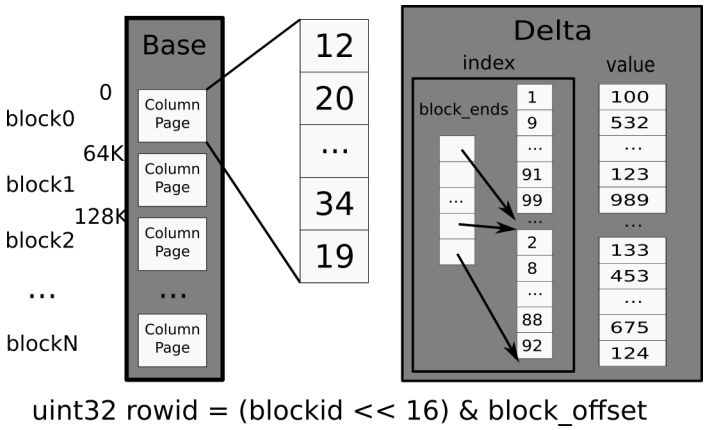

Figure 4: base delta structure

Write pipeline

Each write TX is a large batch of (partial) rows, containing:

- insertion, all columns
- update, key columns and updated columns
- delete, key columns and update on delete flag column

Before supporting update, Doris load task send RPC to BE containing full(all column present) row batches, in order to support update/delete, row batches need to support PartialRow(like in Kudu), each PartialRow only store/serialize key column and updated columns, each PartialRow contains:

- a type field, mark this row is for insert/upsert/update/delele
- a `set` bit vector to mark which column is present
- a `null` bit vector to mark nullable columns' null value
- all valid(set and not null) columns serialized sequentailly

There can be multiple concurrent load tasks write to multiple write Txs, but commit is done serially.

The write TXs need to be serailized and write to WAL(or to a file as `virtual` log entry).

When a TX is committed, it will be applied to main memory storage, converted to Deltas(one for each updated column) and new rows in Base.

Commit version is specified by doris FE.

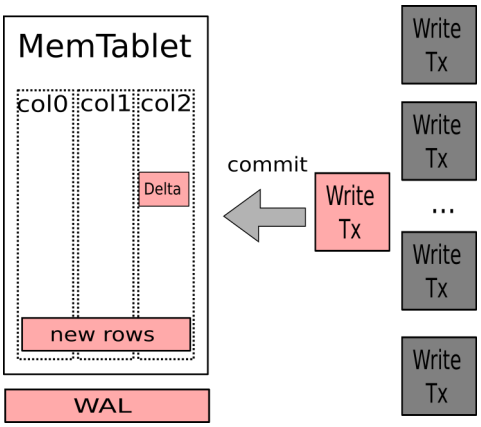

Figure 5: Write TX

```
load task:

tx = talet.create_write_tx
tx.add_row
...
tx.add_row
tx.prepare
tx.commit

talbet service

prepare(tx):
  serialize tx to file or WAL

commit(tx, version):
  write commit msg to WAL
  for row in rows:
    route this row to coresponding SubTablet(currently just one)
    rowid = hashindx.find_or_insert(row.key, next_row_id)
    if is insert:
      next_row_id++
      update total size for this version
      append cells to base
    if is delete:
      add a entry in Delta for delete flag column
    if is update:
      for column, value in row.modified_columns:
        add a entry in Delta for this version
```

The main downside of this design is that commit is done serially, only one thread is doing the heavy `apply` work, which will limit the load throughput, and the `call` to commit will be slow, the considerations behind this are:

1. This is a storage engine for analytical use-case, so most CPU resource should be used for executing queries.

2. `apply` should be fast enough, as mentioned earlier, single-thread load throughput should >100MB/s or 100K rows/s.
3. We can always use more tablets to increase load throught.
4. this could be optimized in the future.

Delta Compaction & memory GC

As `Delta` accumulates, scans/reads need to merge more and more Deltas, delta compaction can `move` Base to a newer version, during compaction, a new Base and Delta will be created, so reads are not blocked, also Delta's index array will not change after compaction, so it can be shared to save memory. The compaction process involves only 1-2 array memory copy and some cell copies, so it should be very fast.

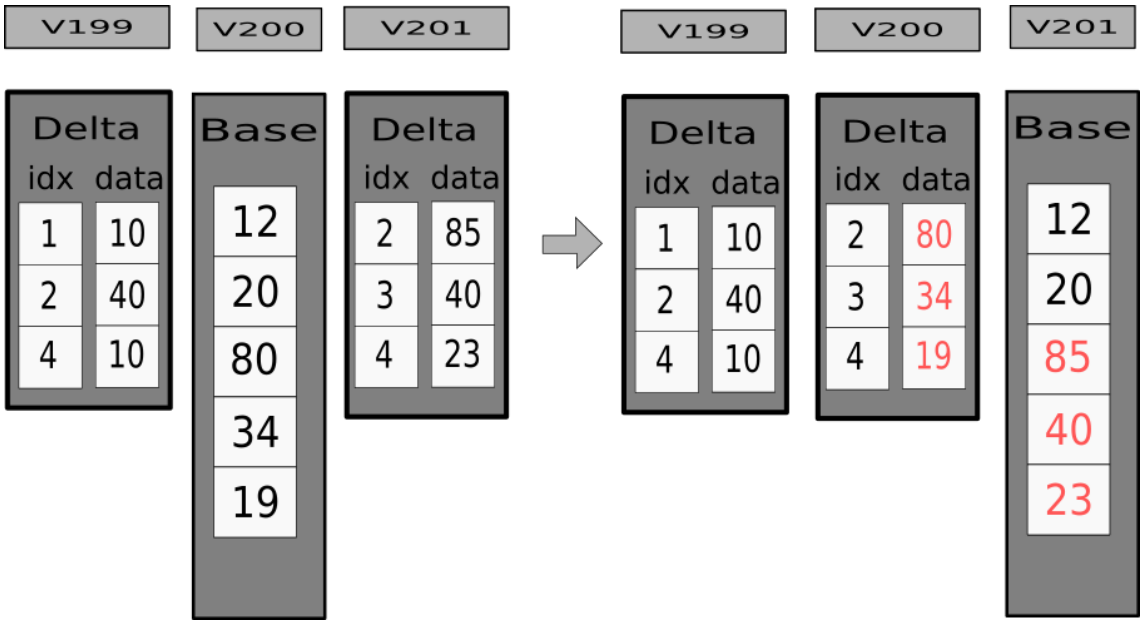

Figure 6: Delta compaction

The engine only keeps Deltas of recent versions(i.e. 5 minutes), old deltas will be GCed to save memory.

Read pipeline

Scan with predicate and aggregate pushdown

For scans, user can choose a specific version or lastest version, if the version is not the Base's version, merge-on-read is performed on related Base and Deltas, merge uses integer rowid offset to apply delta cells, which is much faster than traditional rowkey based join.

Point query

The storage engine also can do fast point queries, to support low selectivity filters. Point query on `Base` is just an array lookup, point query on `Delta` requires a binary search on index array, to further speed-up search, each 64K block can have a 64 byte(1 cache line) index, AVX256 instruction can be used to check 16 uint16_t entries at a time.

Alter schema

- Can alter schema between write TXs(load tasks)

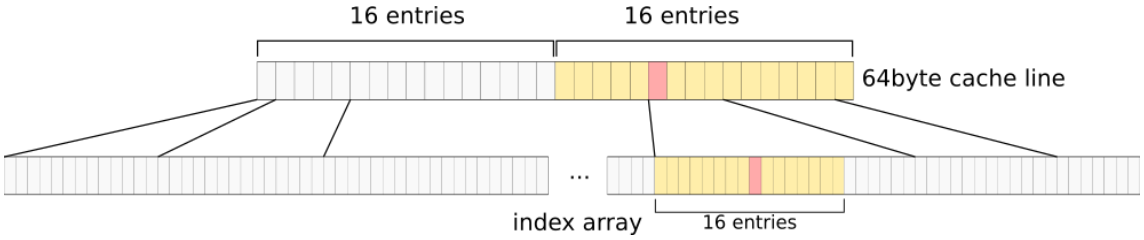

Figure 7: Index SIMD query

- Cannot change key column
- Compatible changes:
  - Can change column name (internally use column id)
  - Can add new column with default value
  - Can delete column
- Incompatible change:
  - change column type (add a new column, copy value from old column to new column, then delete old column)

Snapshot & log GC

Periodically, a snapshot will be flushed to each column's page file on local disk (incrementally, only flush changed pages), this will mix multiple versions of data intertwined together, making it hard to delete old snapshots. After garbage reaches a certain ratio, a full snapshot can be built in the background, then the old page file can be deleted.

Other indexes

Since update is possible, the storage engine can now support various updatable indexes.

For tables that support update, the old rollup table won't work. Instead, rollup index can be used. Rollup index belongs to the tablet and will be updated along with the main tablet.

Other indexes may also be added, and they can be updated along with the main data.

Other systems

Kudu

Kudu is is a distributed `HTAP` supporting massive concurrent read/insert/update/delete while also supporting fast analytical scans, it only support(and excel at) single-row transactions at high qps, at the same time providing fast scans. But if updates are frequent, there well be performance degradation. Kudu is still a `disk based storage`, only newly inserted rows and deltas are kept in memory in a row-based format, and will be flush to disk when suitable.

Kudu uses the idea that a RowSet is freezed after flush, so cells can be accessed using an integer rowid. We also adopt this technique but apply it to the extreme, all the rows in a MemSubTablet reside in a single array(rowid space), the assumption is that memory is sufficient(because only hot data(hash index, key columns and updated columns) needs to be in memory, and maybe PMEM will be popular). Delta compactions are done in-memory per column, so it should be very fast, and it will be done frequently&aggressively. There is a limitation that total row count for a simgle tablet needs to be relatively small(at ~10M range), so user needs to carefully use hash partition(bucket) to keep individual tablet size in certain range.

In-Memory DB

Most In-Memory DB are used for OLTP, which have full transcation support(i.e. multi-statement tx, read-write tx, rollback etc.), and expect highly concurrent small write TXs, a typical OLTP database uses row based model and optimisic MVCC and/or some locking to archieve high concurrency. That's not our targeted use-case.

Analytical column store MPP

Greenplum and Vertica MPP, they expect user to bulk load data using SQL and load statements, also allow small insert/update/delete ops(though have bad performance), have full transcation support, using MVCC and/or some locking, most of those system are designed for bulk insert/load, some have limited update supported, but not designed for frequent updates.