MEP 19 -- Support Plugin KV storage

Current state: Accepted

ISSUE: https://github.com/milvus-io/milvus/issues/11182

PRs:

Keywords: Primary key Deduplication, KeyValue Store, Disaggregate storage and computation

Released: with Milvus 2.1

Authors: Xiaofan Luan

Summary(required)

We want to introduce a global storage to index the relationship between primary key and segment+segment offset.

The index should has the following characteristics

1 Must be able plugin with other cloud native database, such as Dynamo, Aurora, Apache Cassandra, Apache HBase...

2)   Able to scale to at least 100B entites, support 100m+ TPS/QPS

3). Ms level response time

4)   Not introduce another local state to milvus component, it has to be maintained by cloud service or rely mainly on object storage and pulsar.

5)   Query strong consistent data under certain Ts.

The index can be further extended to support global secondary index, as well as Full-text index in the future.

Motivation(required)

See git issue:

#10712
#7130

In many situations, Milvus need an efficient way to find the primary key location(Which Segment the primary key is in).

For example, we may want to dedup by primary key, in the write path we will need to know if the pk is already written in Milvus or not, Another case is when we want to find which segment the pk is in while execute delete, so far the delete use BloomFilter to filter out segments not related with this primary key, however the false positive may cause unnecessary delete delta logs in sealed segments.

Certain Query/Search request can also utilize the KV index. First step is to support query by pk request, it could also be used for other field indexing later on. For instance, if a collection has 3 field:

1) pk a int64 2) b string 3) c vector. The KV index should be able to help on the following Query:

Query xxx where a = 1

Query xxx where a > 1 && a < 3 (That might be not efficient if we partition the kv index by range)

KV Storage can also be used to lookup the reduced result. Currently we will have to retrieve the whole entity from Segcore after local reduce and sent back to proxy, there are two problems of current design

1) it K is large and users fetch many fields, the query results will be really large, while most of the query results are not really necessary and will be dropped under global reduce.

2)all the fields in segcore has to be loaded into memory, the vector field might be too large to fit into memory. we've already support local caching, but all the storage is under columnar mode, a row based storage can better serve retrieve by id under some situations.

Public Interfaces(optional)

KV interface

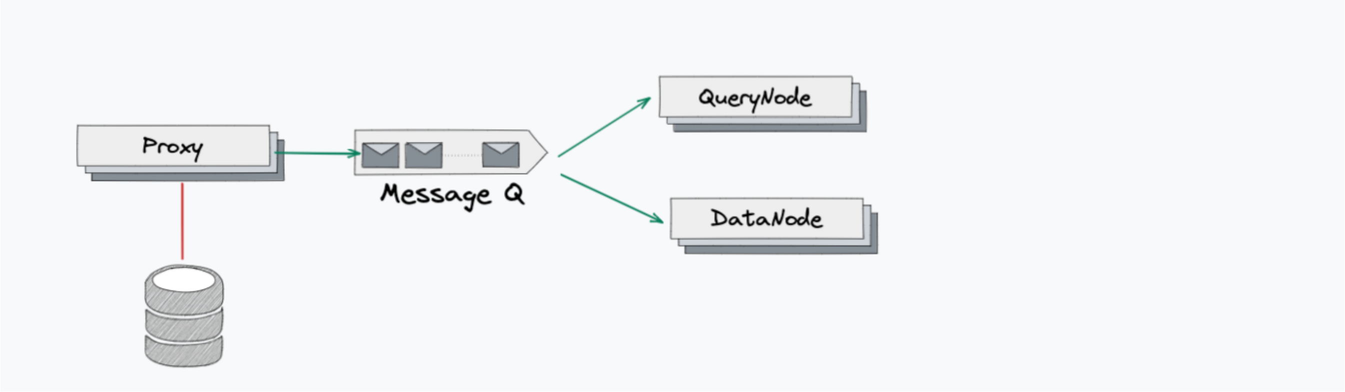
```
Put(key, value string) error
BatchPut(kvs map[string] string) error
Delete(key string) error
BatchDelete(keys [] string)
DeleteRange(start string, end string) error
DeletePrefix(prefix string) error

Get(key string) string
BatchGet(key string[]) string[]
Scan(startKey string, endKey string) string[]
ScanPrefix(prefixKey string) string[]

//Read modify operation
Merge(key, value string)
// most of the interface should be extended from rocksdb
```

Design Details(required)

1) Basic work flow



Plan1

Proxy directly interact with KV Index.

Pros : No need to worry about consistency, read always the lastest data.

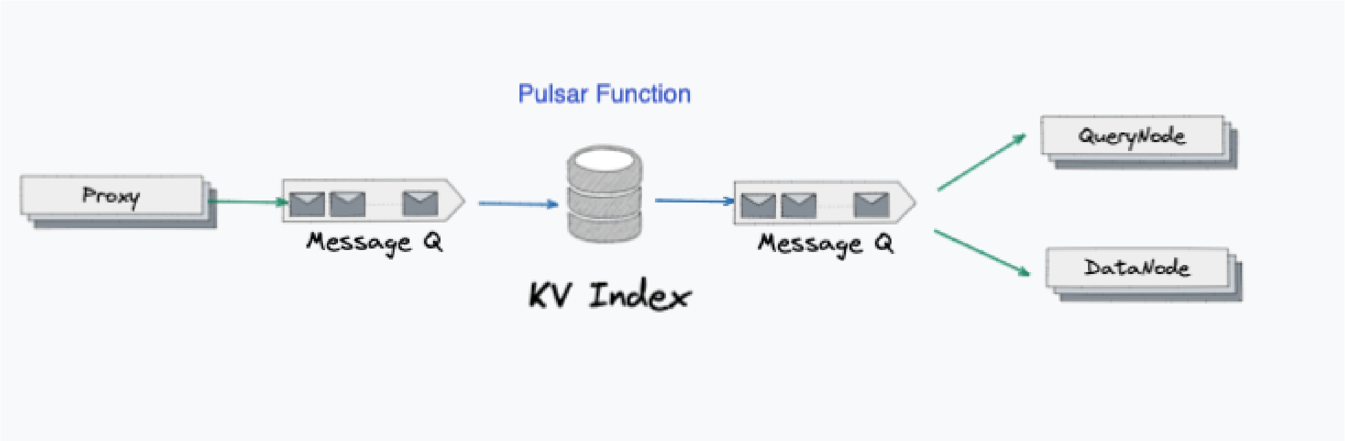
lookup reduced result is easy.

if write duplicated PK, proxy can directly return error to client.

Easy to implement

Cons: Hard to maintain the consistency between KV storage and message queue, no single source of truth

KV storage has to handle data persistence itself.



Plan2

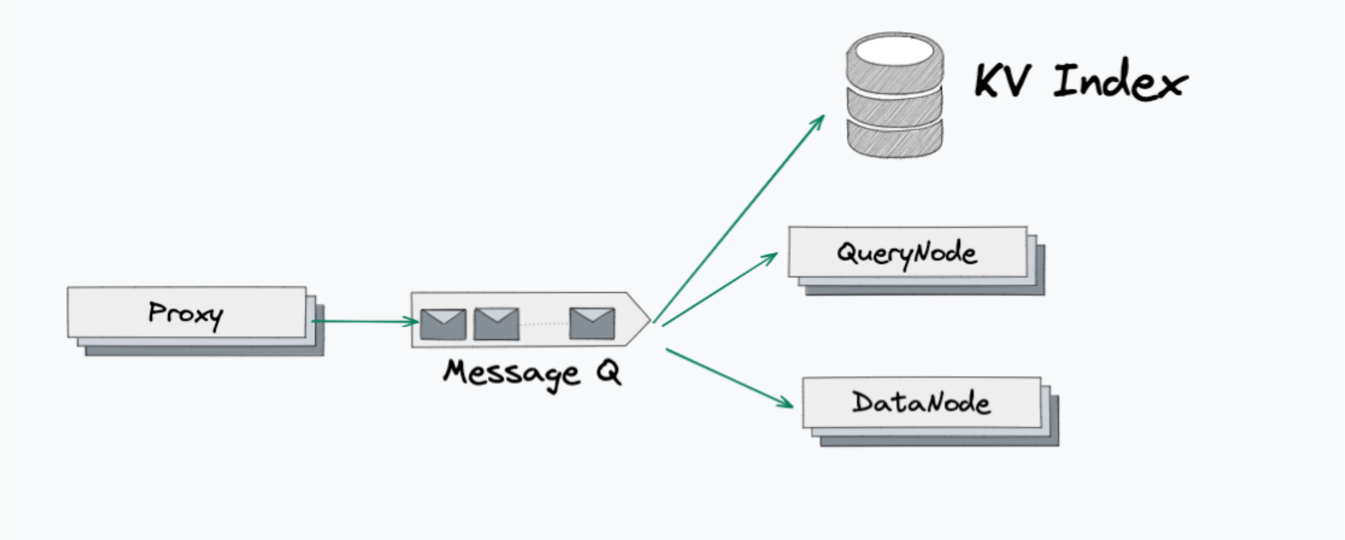
Proxy directly write into message queue, utilize pulsar function to query and update KV index about which segment the delete belongs to, and write back the modifed binlogs into second stage pulsar.

Pros: The logic is straight forward, query and data node logic doesn't need to change

No need to worry about consistency, read always the lastest data. form KV index.

Cons Couple with pulsar too much, hard to change pulsar to other message storage, especially cloud message storages such as Kenesis.

Performance will degrade to half because we write message queue twice.



Plan3

KV index works as a log consumer. implements KV node and consumes data from message queue.

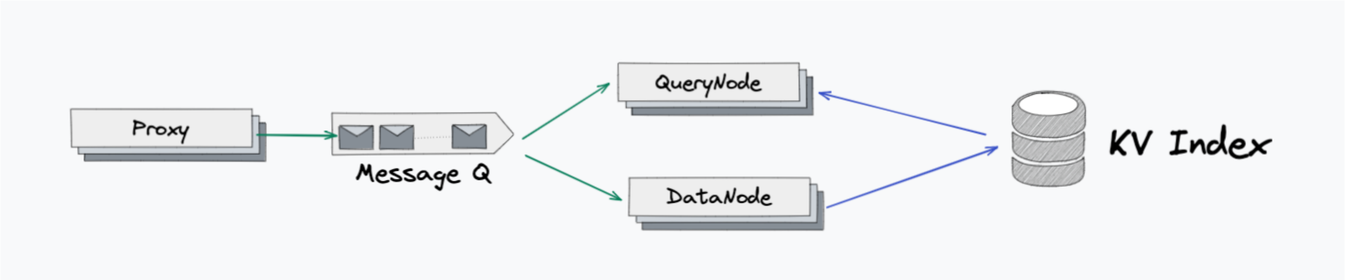
Pros: System architecture looks clean and easy to understand

Pulsar/Kafka connector helps to convert bin log and write into KV index.

could handle failure recovery by replaying message Q, but might be very slow since the kv index has different sharding policy with milvus sharding policy

Cons: each DML need to check from the KV index about the duplication, which might slow the system overall performance.

The schedule code of KV node might be very similar to querynode, reuse the same logic?



Plan4

DataNode response for written KV index, QueryNode maintains incremental KV data in memory, once Datanode flush to S3, Querynode load historical KV index from S3 and serves read.

Pros: Performance loss is minimal

could handle failure recovery by replaying message Q, but might be very slow since the kv index has different sharding policy with milvus sharding policy

Follow the design rules of datanode write data and querynode serve data.

Cons: Complexity is high, but might reuse current segment replace logic.

Might need to do rpc between querynode querynode(for deduplication) and proxy querynode(for retrieve entities).

Need have a discussion about whether plan3 or plan4 is favored.

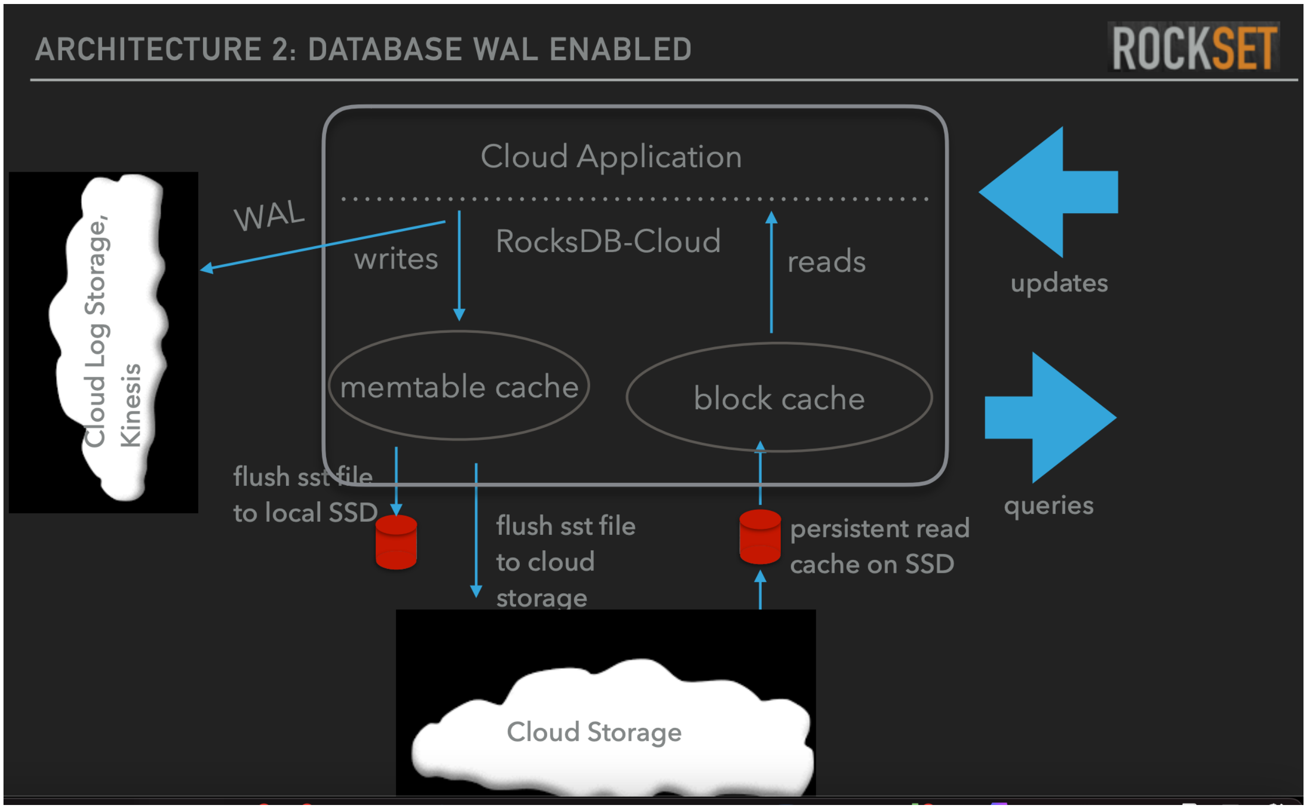
2)What KV engine to support

we will support plugin storages, most of the storage are maintained by cloud provider or user's exist deployment. However, we still need one default implementation.

To achieve storage/computation disaggregation, the storage engine must support to store data on S3/Minio and other object storage, we find rocksdb-cloud, which could tail data easily from distributed log storage, store sstable on object storage, and serve read/write in miliseconds.

if you are interested in the rocksdb cloud, please take a look at http://borthakur.com/ftp/rocksdb-cloud_dhruba_borthakur.pdf

blocked URL

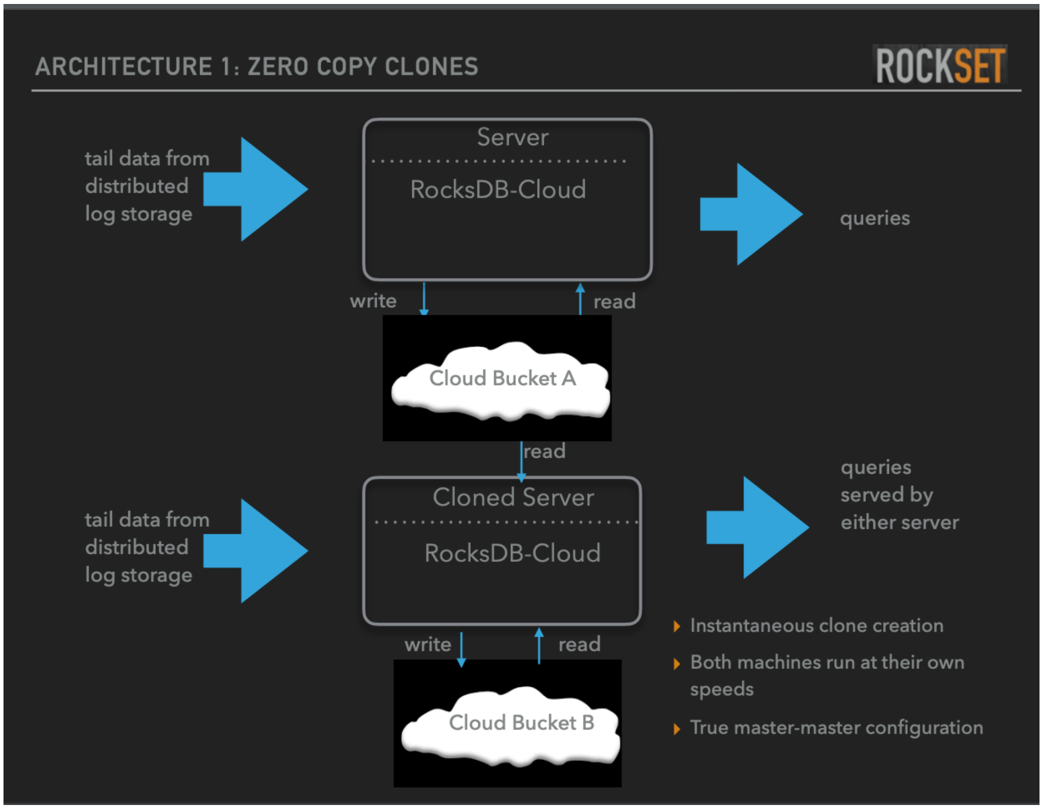


ARCHITECTURE 2: DATABASE WAL ENABLED

ROCKSET
Cloud Application
RocksDB-Cloud
WAL
writes
reads
updates
queries
memtable cache
block cache
Cloud Log Storage, Kinesis
flush sst file to local SSD
flush sst file to cloud storage
persistent read cache on SSD
Cloud Storage

architecture of rocksdb cloud

Better feature of rocksdb cloud is Zero copy clones, this feature perfectly matches the way datanode and querynode works. datanode convert binlog into sstables while flush, write data into cloud bucket and notify querynode, while querynode can simply load data from S3 and remove theird local memtables.



ARCHITECTURE 1: ZERO COPY CLONES

ROCKSET

tail data from distributed log storage

Server

RocksDB-Cloud

queries

write

read

Cloud Bucket A

read

tail data from distributed log storage

Cloned Server

RocksDB-Cloud

queries served by either server

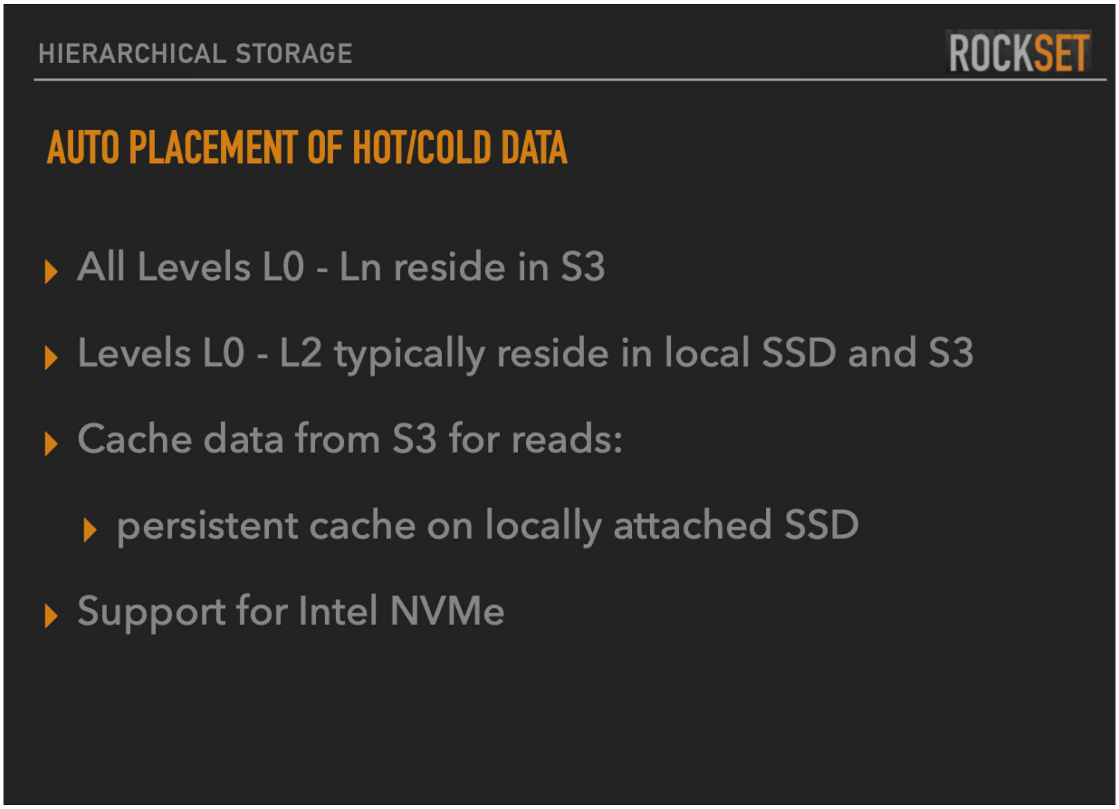
write

read

Cloud Bucket B

- Instantaneous clone creation
- Both machines run at their own speeds
- True master-master configuration

Rocks Cloud also support tiered storage, we can caching hot data in local SSD while S3 is used for durability



HIERARCHICAL STORAGE

ROCKSET

AUTO PLACEMENT OF HOT/COLD DATA

- All Levels L0 - Ln reside in S3
- Levels L0 - L2 typically reside in local SSD and S3
- Cache data from S3 for reads:
  - persistent cache on locally attached SSD
- Support for Intel NVMe

How to shard data and route

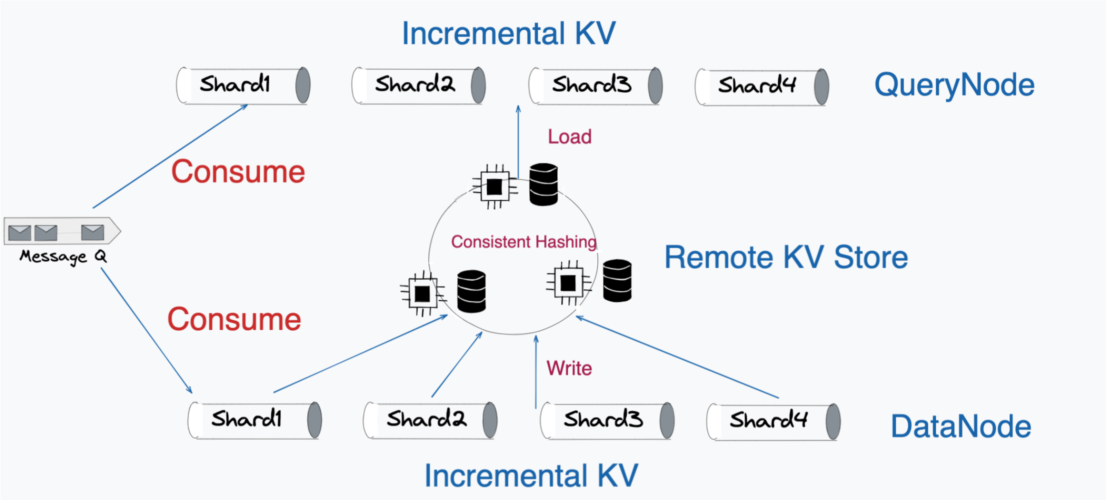
To serve billlions of data, we need to shard the kv storage for distribution

1. same shard as milvus itself if shard number is not small, might block kv read and can not split
2. shard by range
3. shard by hash

Consistent Hashing is better balanced, easier to implementation, but range operation on parition key is slower.

Range partition requires subtle operation to split and merge exist partition to maintain load balance(Can not allocate shards beforehand unless we know the key distrbution in advance otherwise it will not be balanced)

From our use case, point queries are called more frequently, thus consistent hashing seems to fit our requirement.



The final KV Storage is illustrate in the above graph. QueryNode and DataNode both hold cloud rocksdb instance. Data is shared between two instance through the zero copy clone feature of rocksdb cloud.

Like Milvus, we separate data into two parts, incremental and historical. Incremental data only exists in querynode, and only exist for growing segments. Datanode consumes binlog from message stream, and write directly into remote KV store, once the segment is sealed, we can remove the caching at querynode side since we know all primary key of this segment has already been consumed by datanode and written into the remote KV store.

For deduplication and delete workloads, datanode can query from remote KV store directly because keys are consumed from message storage in order, thus all previous keys must already be visible at remote KV store.

For querynode, some of the keys may not be in remote kv store yet because data and query node may have different consume speed. We will need to query incremental KV in querynode memory to cover the latest write ins.

Disaster recovery

1) If Data node crash, not even need to do anything because current datanode recovery will handle the message queue reconsuming and write back.

2)if query node crash, will need to consume from message stream and rebuild incremental kv in memory

3)if remote KV store crash 1.shard should be taken over by other kv stores.2. sstables can be directly loaded from object storage.

3.memtable need to be restored from message stream, which maybe slow because we need to filter out most majority of the data.

if KV store rewrite the log to message storage again the failure recovery should be simple and fast, but it requires extras IOs (prefered)

Test Plan(required)

1) Test the RocksDB store on pulsar and S3, verify it works

2) Test consistent hash policy and remote access

3) Integrate KV storage with Milvus

4) Integrate external KV storage with Milvus

Rejected Alternatives(optional)

1. Use Query Api to find primary key, it need to query in each segment thus not efficient enough.
2. Use bloom filter index to filter the irrelevant segments

3. same shard policy between milvus and kv store?