Architecture of KingDB v0.9.0

Legend

Thread

Event notifier

Method or function

Comment or explanation

1. ... 11. Step-by-step overview of the write process

1. ... 5. Step-by-step overview of the read process

Client Application

Client #1 ... Client #K

The client threads execute Put(), Get() and Delete(), therefore they are the ones performing compression and checksum for the entries they handle. With this design, the CPU workload is spread across the client threads.

1. *Client #1 calls the Put() method of KingDB*

1. *Client #K calls the Get() method of KingDB*

Database

Put() Delete() Get()

Snapshot

Iterator

Multipart API

2. *The interface passes the data to the PutPart() method of the Write Buffer*

All Delete() must be persisted to disk, therefore they are all turned into writes just like calls to Put()

Write Buffer

PutPart() Get()

3. *Incoming Put() and Delete() operations are written to the incoming buffer.*

Incoming Buffer
std::vector<kdb::Order>

Flush Buffer
std::vector<kdb::Order>

Buffer Manager

2. *Get() checks the Write Buffer first in case the entry was just inserted and has not been persisted yet*

3. *If the entry is not in the Write Buffer, Get() checks the Storage Engine by calling GetEntry()*

Unlike the Put() and Delete() that are buffered, the Get() accesses the Storage Engine directly

4. *When the Incoming Buffer is full, the Buffer Manager thread swaps it with the Flush Buffer and passes it to the Event Manager. The Incoming Buffer continues to take incoming writes while the Flush Buffer is being written to disk*

11. *The Write Buffer is notified that the Storage Engine is done writing the buffer and updating the index, and is ready to handle the next buffer*

Event Manager

Flush Start event | Index Update event | Flush End event

The Write Buffer, Storage Engine and Index are synchronized using a set of events built on top of a messaging library the Event Manager. This reduces overall architectural dependencies.

5. *The Flush Start event notifies the Storage Engine that a buffer is ready to be flushed*

8. *The Storage Engine notifies that the buffer has been written*

9. *The Index Updater thread now needs to reference the entries from the buffer into the index*

Storage Engine

GetEntry()

Entry Writer

6. *The Entry Writer thread passes the buffer to the HSTable Manager*

System Statistics Poller

Compactor

Index Updater

10. *The Index Updater notifies back once it has referenced all entries*

4. *GetEntry() queries the index to find the location of the entry using its key.*

HSTable Manager

WriteOrdersAndFlushFiles()

7. *The parts of entries in the buffer are written to the file system in HSTable format*

Index

(in-memory hash tables)

Main index
std::multimap<uint64_t, uint64_t>

Temporary index
std::multimap<uint64_t, uint64_t>

The temporary index has the same structure as the main index, and is used when the primary index needs to be read-only (ex: when compaction is running)

File system

(files stored on disk)

/path/db/file1 ... /path/db/fileN

Writes are done with calls to pwrite(). Reads are done through read-only memory maps.

Data is never overwritten, and always written to a new file. Large entries have their own dedicated files.

5. *Once the location is found, the entry is retrieved by accessing the file system directly.*

KingDB uses various classes to perform utility tasks, data processing and parametrization. Because these classes are used everywhere, their interactions in the architecture have not be represented in this diagramThey are listed below as a reminder that they all are important building blocks of the KingDB architecture.

Utilities

Status

ByteArray

Order

Logger

FileUtil

Algorithms

Compressor

Hash

Checksum

Parametrization

DatabaseOptions

ReadOptions

WriteOptions

ServerOptions