

Architecture of KingDB v0.9.0

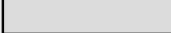
Legend



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

Client #1 calls the Put() method of KingDB

1.

Client #K calls the Get() method of KingDB

1.

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.

Database

Put()

Delete()

Get()

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

2.

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

Write Buffer

PutPart()

Get()

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

3.

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

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

Buffer Manager

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

4.

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

11.

Event Manager

Flush Start event

Index Update event

Flush End event

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

5.

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

Entry Writer

System Statistics Poller

Compactor

GetEntry()

The Entry Writer thread passes the buffer to the HSTable Manager

6.

The Index Updater notifies back once it has referenced all entries

10.

HSTable Manager

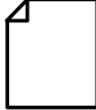
WriteOrdersAndFlushFiles()

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

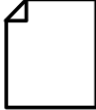
7.

File system

(files stored on disk)



...



/path/db/file1

/path/db/fileN

Index

(in-memory hash tables)

Main index
std::multimap<uint64_t, uint64_t>

Temporary index
std::multimap<uint64_t, uint64_t>

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

4.

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)

5.

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

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.

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 been represented in this diagram. They 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