IOHK | SCOREX BLOG

SCOREX BLOG

IODB storage engine

OCTOBER 11, 2016 JAN KOTEK 8 MIN READ COMMENTS

Log-Structured-Merge trees (LSMT) are a good fit for modern SSD storage and offer good performance and reliability. LSMT are also a good fit for blockchain storage requirements (snapshots, consistency, proof of existence). This blog post describes a database designed specifically for blockchain storage, inspired by existing LSMT implementations (RocksDB, COLA tree).

The current state-of-the-art LSMT implementation is probably RocksDB, with in-memory write buffers, parallel compaction and snapshots. Another similar algorithm are COLA tree. That is a btree-like structure where each node has separate write buffer. Finally there is SSTable from Cassandra which is fairly simple, but offers great write performance.

For Scorex and other projects, we designed a storage with codename IODB, which combines features from all the above implementations. It has log-structured storage with fast binary search and multilevel-multithreaded compaction. It will also have snapshots, MVCC isolation, concurrent transactions, online backups etc…

Compared to RocksDB IODB it will have better concurrency, lesser write amplification and smaller disk overhead during compaction. IODB also runs on JVM, is easier to deploy and its functionality can be extended with plugins.

RocksDB trade-offs

RocksDB is probably the current state-of-the-art LSMT implementation. It offers great performance and great concurrency. However it also has some trade-offs.

RocksDB (and other LSMT implementations) is using compaction to reclaim disk space, and to remove the old version of data from logs. The compaction process is not optimal:

- To perform full compaction all files have to be rewritten.
- Full compaction requires extra disk space to create new files, twice more than the data size.
- Compaction process operates over large files. It can take a long time before a single compaction task finishes. This makes it hard to temporarily pause the compaction process or close the database quickly.
- Write amplification is too big, the data entries are moved too many times during compaction process,

even if they were not modified.

IODB Design

To solve that IODB took an inspiration from COLA tree (also known as Fractal Tree), which offers great performance under random updates. A COLA tree is a BTree like structure, where each node has its own write buffer. If the buffer becomes too large its content is propagated to a lower level by compaction process. A COLA tree is complex, but the basic idea of separate writer buffers is simple and great.

A COLA tree is difficult to implement; the code for compaction and rebalancing is complex, almost impossible in a concurrent environment. IODB avoids that by sharding data into non-overlapping intervals, rather than hierarchical BTree like nodes. Self-balancing code is simple; if an interval becomes too large (over 64MB), it is sliced into smaller intervals by the compaction process. If an interval becomes too small, it is merged into its neighbours.

Each interval in IODB is compacted with multi-level merge, the same way as RocksDB or any other LSMT implementation. So in practice IODB is composed of several small LSTM, one for each interval.

Background compaction is composed of several small atomic tasks. Each task operates on a single interval with limited size (bellow 64MB). Small tasks are easier to tune and run better in multiple threads. The limited size requires very little space overhead. Finally it is fast to close store or temporary pause compaction.

Multi source pre-sorted merge

Every commit (or write batch) is saved into a separate file. Over time you get multiple files sorted by time. To find an entry one starts from the newest file and traverses all files, until the required key is found. The compaction process merges those multiple files together.

RocksDB compaction can merge only two files at a time. IODB compaction can merge more than two files at a time. If the source data are presorted, the multi merge requires only one single pass over the source data. Creating the new merged file takes approximately `o(N*log(M))` time, where N is the total number of entries in interval, and M is the number of source files.

The number of source files (M) and the interval size (N) can be tuned. Large M means less IO and more CPU usage. Larger N reduces IO and CPU usage at the expense of larger memory usage. Configuring those parameters should make IODB usable in various workloads.

There is a number of optimizations. For example to reduce CPU overhead from comparing keys, IODB can be configured so that all keys in one interval share the same prefix. Only the key suffix is then compared (this trick comes from Cassandra).

Binary search over sorted table

Scorex uses fixed size keys. We can eliminate a file offset translation layer, and perform binary search directly over a memory-mapped file. This is very disk cache friendly and much faster than traditional hierarchy based structures, such as BTrees.

Deployment

IODB is designed to be very easy to deploy. It is pure-Java and can run on any Java8 enabled JVM.

It is also written in a way which allows to integrate it into J2EE container or Spring Dependency Injection. It is a set of independent components wired together with the constructor parameters.

It also uses common Java components such as `ScheduledExecutorService` to run background tasks. IODB can share thread pools with other Java libraries etc..

Finally IODB takes some features from RocksDB. It will support incremental backups and snapshots, created with hard file links…

MapDB implementation

MapDB will not use IODB directly, but will get a storage component based on IODB design. MapDB needs a different set of features (variable key size, 8-byte recids, less random data) and is written in a different language (Scala vs Kotlin), so it will have to get separate implementation.

MapDB will get append-only storage engine based on IODB design. Recids (keys) in MapDB store are 8-byte longs, which allows many optimizations by using primitive arrays and collections.

`SortedTableMap` could also use compaction similar to that of IODB design. So we will have a writable `NavigableConcurrentMap` on top of the append-only storage. It will use `SortedTableMap` storage format for a single file, but it will support updates, snapshots, compaction and all the features of IODB.

October 11, 2016 by jan.kotek@iohk.io

IODB storage engine

Log-Structured-Merge trees (LSMT) are a good fit for modern SSD storage and offer good performance and reliability. LSMT are also a good fit for blockchain storage requirements (snapshots, consistency, proof of existence). This blog post describes a database designed specifically for blockchain storage, inspired by existing LSMT implementations (RocksDB, COLA tree).

May 17, 2016 by alex.chepurnoy@iohk.io

Announcing Ergaki - A performant, public bulletin board for voting and auctions

The first Scorex-based testnet, Lagonaki, combines the Permacoin consensus protocol implementation with a simple, Nxt-like payments module. After Lagonaki, the next Scorex-based testnet will be *Ergaki*, a block chain system that will be used as a public and performant bulletin board for various protocols including voting and auctions.

May 17, 2016 by alex.chepurnoy@iohk.io

Ergaki, the Next Scorex Testnet

A Scorex application is comprised if core, and Scorex itself is the core functions and module interfaces, and modules. The current testnet, Lagonaki, is made of Permacoin consensus protocol implementation and a simplest Nxt-like payments module.