MapDB

Release 2.0

Jan Kotek

July 23, 2015

CHAPTER

ONE

FOREWORD

MapDB is an embedded database engine for java. It provides collections backed by on-disk or in-memory storage. It is flexible, simple and free under Apache 2 license. MapDB is extremely fast, as in-memory mode rivals on-heap `java.util` collections and on-disk mode outperforms databases written in C. And it has several configuration options from regular ACID transactions to very fast direct storage.

MapDB started in 2001 under the name JDBM. The current MapDB2 is its 5th generation. It evolved as a db engine focused around Java. It is pure-java, implements Maps, has only single 500KB jar, and runs everywhere, including Android and JDK6. Also, its serialization lifecycle and separate memory allocator makes it easy to port existing data models (such as collections) to database engine, outside of Garbage Collector influence.

This manual is a work in progress and it will be completed together with the MapDB 2.0.0 release. I hope you will find it useful. I would be very happy to accept pull requests

CHAPTER

TWO

GETTING STARTED

MapDB has very power-full API, but for 99% of cases you need just two classes: DBMaker is a builder style factory for configuring and opening a database. It has a handful of static 'newXXX' methods for particular storage mode. DB represents storage. It has methods for accessing Maps and other collections. It also controls the DB life-cycle with commit, rollback and close methods.

The best places to checkout various features of MapDB are Examples. There is also screencast which describes most aspects of MapDB.

There is `MapDB Cheat Sheet`, a quick reminder of the MapDB capabilities on just two pages.

2.1 Maven

MapDB is in Maven Central. Just add the code bellow to your pom file to use it. You may also download the jar file directly from repo.

```
<dependency>
    <groupId>org.mapdb</groupId>
    <artifactId>mapdb</artifactId>
    <version>1.0.8</version>
</dependency>
```

We are working on the new generation of MapDB. It is faster and more reliable. The latest semi-stable build is at snapshot repository:

```
<repositories>
    <repository>
        <id>sonatype-snapshots</id>
        <url>https://oss.sonatype.org/content/repositories/snapshots</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>org.mapdb</groupId>
        <artifactId>mapdb</artifactId>
        <version>2.0.0-SNAPSHOT</version>
    </dependency>
</dependencies>
```

2.2 Hello World

Hereafter is a simple example. It opens TreeMap backed by a file in temp directory. The file is discarded after JVM exits:

```
// import org.mapdb.*;
ConcurrentNavigableMap treeMap = DBMaker.tempTreeMap();

// and now use disk based Map as any other Map
treeMap.put(111,"some value");
```

This is a more advanced example, with configuration and write-ahead-log transaction.

```
// import org.mapdb.*;

// configure and open database using builder pattern.
// all options are available with code auto-completion.
DB db = DBMaker.fileDB(new File("testdb"))
        .closeOnJvmShutdown()
        .encryptionEnable("password")
        .make();

// open existing an collection (or create new)
ConcurrentNavigableMap<Integer,String> map = db.treeMap("collectionName");

map.put(1, "one");
map.put(2, "two");
// map.keySet() is now [1,2]

db.commit();  //persist changes into disk

map.put(3, "three");
// map.keySet() is now [1,2,3]
db.rollback(); //revert recent changes
// map.keySet() is now [1,2]

db.close();
```

2.3 What you should know

MapDB is very simple to use, however it bites when used in the wrong way. Hereis a list of the most common usage errors and things to avoid:

- Transactions (write-ahead-log) can be disabled with DBMaker.transactionsDisable(), this will speedup the writes. However, without WAL the store gets corrupted when not closed correctly.
- Keys and values must be immutable. MapDB may serialize them on background thread, put them into instance cache... Modifying an object after it was stored is a bad idea.
- MapDB relies on memory mapped files. On 32bit JVM you will need DBMaker.randomAccessFileEnable() configuration option to access files larger than 2GB. RAF introduces overhead compared to memory mapped files.
- MapDB does not run compaction on the background. You need to call `DB.compact()` from time to time.
- MapDB uses unchecked exceptions. All `IOException` are wrapped into unchecked `IOError`. MapDB has weak error handling and assumes disk failure can not be recovered at runtime. However, this does not affect the

data safety, if you use durable commits.

CHAPTER

THREE

DB AND DBMAKER

MapDB is a set of loosely coupled components. One could instantiate components manually with contructors and parameters. To make things easier there are two factory classes to do: `DBMaker` and `DB`. They use maker (builder) pattern, so most configuration options are quickly available via code assistant in IDE.

DBMaker handles database configuration, creation and opening. MapDB has several modes and configuration options. Most of those can be set using this class.

DB represents opened database (or single transaction session). It creates and opens collections . It also handles transaction with methods such as `commit()`, `rollback()` and `close()`.

To open (or create) a store use one of `DBMaker.xxxDB()` static methods. MapDB has more formats and modes, each `xxxDB()` uses different mode: `memoryDB()` opens in-memory database backed by `byte[]`, `sppendFileDB()` opens db which uses append-only log files and so on.

`xxxDB()` method is followed by configuration options and `make()` method which applies all options, opens storage and returns `DB` object. This example opens the file storage with encryption enabled:

```
DB db = DBMaker
        .appendFileDB(new File("/some/file"))
        .encryptionEnable("password")
        .make();
```

Once you have DB you may open a collection or other record. DB has two types of factory methods:

`xxx(String)` takes the name and opens the existing collection (or record), such as `treeMap("customers")`. If a collection with the given name does not exist, it is silently created with default settings and returned. An example:

```
NavigableSet treeSet = db.getTreeSet("treeSet");
```

`xxxCreate(String)` takes the name and creates a new collection with customized settings. Specialized serializers, node size, entry compression and so on affect performance a lot and they are customizable here.

```
Atomic.Var<Person> var = db.atomicVarCreate("mainPerson", null,  Person.SERIALIZER);
```

Some `create` method may use builder style configuration. In that case you may finish with two methods: `make()` creates a new collection and if a collection with the given name already exists, it throws an exception. Method `makerOrGet()` is the same, except if a collection already exists, it does not fail, but returns the existing collection.

```
NavigableSet<String> treeSet = db
        .treeSetCreate("treeSet")
        .nodeSize(112)
        .serializer(BTreeKeySerializer.STRING)
        .makeOrGet();
```

3.1 Transactions

`DB` has methods to handle a transaction lifecycle: `commit()`, `rollback()` and `close()`.

```
ConcurrentNavigableMap<Integer,String> map = db.getTreeMap("collectionName");

map.put(1,"one");
map.put(2,"two");
//map.keySet() is now [1,2] even before commit

db.commit();  //persist changes into disk

map.put(3,"three");
//map.keySet() is now [1,2,3]
db.rollback(); //revert recent changes
//map.keySet() is now [1,2]

db.close();
```

One `DB` object represents single transaction. The example above uses single global transaction per store, which is sufficient for some usages.

Concurrent transactions are supported as well, with full serializable isolation, optimistic locking and MVCC snapshots. For concurrent transactions we need one extra factory to create transactions: `TxMaker`. We use `DBMaker` to create it, but instead of `make()` we call `makeTxMaker()`

```
TxMaker txMaker = DBMaker
        .memoryDB()
        .makeTxMaker();
```

A single `TxMaker` represents an opened store. `TxMaker` is used to create multiple `DB` objects, each representing a single transaction. In that case `DB.close()` closes one transaction, but storage remains open by `TxMaker`:

```
DB tx0 = txMaker.makeTx();
Map map0 = tx0.treeMap("testMap");
map0.put(0,"zero");

DB tx1 = txMaker.makeTx();
Map map1 = tx1.treeMap("testMap");

DB tx2 = txMaker.makeTx();
Map map2 = tx1.treeMap("testMap");

map1.put(1,"one");
map2.put(2,"two");

//each map sees only its modifications,
//map1.keySet() contains [0,1]
//map2.keySet() contains [0,2]

//persist changes
tx1.commit();
tx2.commit();
// second commit fails  with write conflict, both maps share single BTree node,
// this does not happen on large maps with sufficient number of BTree nodes.
```

TODO snapshots and native snapshots

CHAPTER

FOUR

DURABILITY AND SPEED

There are several configuration options to make compromises between durability and speed. You may choose consistency, disk access patterns, commit type, flush type and so on.

4.1 Transactions disabled

If a process dies in the middle of write, storage files might become inconsistent. For example the pointer was updated with a new location, but new data were not written yet. For this MapBD storage is protected by write-ahead-log (WAL) which applies commits in atomic fashion. WAL is reliable and simple, and is used by many databases, such as Posgresql or MySQL.

However WAL is slow, data has to be copied and synced multiple times. You may optionally disable WAL by disabling transactions: `DBMaker.transactionDisable()`. In this case you **must** correctly close the store before JVM shutdown, or you loose all your data. You may also use a shutdown hook to close the database automatically before JVM exits, however this does not protect your data if JVM crashes or is killed:

```
DB db = DBMaker
        .memoryDB()
        .transactionDisable()
        .closeOnJvmShutdown()
        .make();
```

Transaction disable (also called direct mode) will apply all the changes directly into storage file. Combine this with in-memory store or mmap files and you get very fast storage. Typical use is in a scenario where the data does not have to be persisted between JVM restarts or can be easily recreated: off-heap caches.

Other important usage is for the initial data import. Transactions and no-transactions share the same storage format (except optional write-ahead-log), so one can import data very quickly with transactions disabled. Once the import is finished, the store is reopened with transactions enabled.

With transactions disabled you loose rollback capability, `db.rollback()` will throw an exception. `db.commit()` will have nothing to commit (all data are already stored), so it does the next best thing: Commit tries to flush all the write caches and synchronizes the storage files. So if you call `db.commit()` and do not make any more writes, your store should be safe (no data loss) in case of JVM crash.

4.2 Memory mapped files (mmap)

MapDB was designed from ground to take advantage of mmap files. However mmap files are limited to 4GB by addressing limit on 32bit JVM. Mmap have a lot of nasty effects on 32bit JVM (it might crash if addressing limit is exceeded), so by default we use a slower and safer disk access mode called Random-Access-File (RAF).

Mmap files are much faster compared to RAF. The exact speed bonus depends on the operating system and disk case management, but is typically between 10% and 300%.

Memory mapped files are activated with `DBMaker.mmapFileEnable()` setting.

One can also activate mmap files only if a 64bit platform is detected: `DBMaker.mmapFileEnableIfSupported()`.

Mmap files are highly dependent on the operating system. For example, on Windows you cannot delete a mmap file while it is locked by JVM. If Windows JVM dies without closing the mmap file, you have to restart Windows to release the file lock.

4.3 CRC32 checksums

You may want to protect your data from disk corruption. MapDB optionally supports CRC32 checksums. In this case, each record stores an extra 4 bytes which contains its CRC32 checksum. If the data are somehow modified or corrupted (file system or storage error), the next read will fail with an exception. This gives an early warning and prevents db from returning the wrong data.

CRC32 checksum has to be calculated on each put/modify/get operation so this option has some performance overhead. Write-ahead-log uses CRC32 checksum by default.

Checksum is activated by this setting: `DBMaker.checksumEnable()`. It affects the storage format, so once activated you always have to reopen the store with this setting. Also checksum can not be latter activated once the store was created without it.

TODO: CRC32 serializer link

TODO: WAL and CRC32 disable

TODO: WAL flush on commit

TODO async file sync, use futures?

CHAPTER

FIVE

CACHES

MapDB has several options to cache deserialized objects. A proper cache configuration is crucial for good performance of your application. Many performance problems can be fixed just by changing the cache settings.

Most dbs and the old generation of MapDB (JDBM) use fixed size to cache read from the disk. MapDB eliminated the page layer, so it does not have regular cache. Instead it used memory mapped files and relies on the operating system to do disk caching.

When we talk about cache in mapdb we mean *instance cache* .Instead of pages, MapDB caches deserialized objects, such as tree nodes, `Long`, `Person` and so on. Instance cache helps to minimize deserialization overhead. When you fetch an object twice, it will be deserialized only once. The second time it will be fetched from cache.

Because the object instance may be stored in cache, yor data-model has to be immutable. On every modification you must create a copy of the object and persist the new version. You also can not modify an object fetched from the db. An example of this is the following:

```
//wrong
Person person = new Person();
map.put("John", person);
person.setName("John");

//right
person = new Person();
person.setName("John");
map.put("John", person);

//wrong
person = map.get("John");
person.setAge(15);

//right, create copy which is modified and inserted
person = map.get("John");
person = person.clone(); //defensive copy
person.setAge(15);
map.put("John", person);
```

MapDB offers 5 cache implementations. Some are unbounded and could cause `OutOfMemoryError` when used incorrectly.

5.1 Hash Table cache

This cache is a fixed size hash table (an array), where elements are placed by recid hash. Old entries are evicted by hash collisions. It has almost zero performance overhead and it provides good results, so this cache is recommended.

It does not have an automatic entry eviction, so some records may remain in cache (on heap) for a very long time. This cache should be used in combination with small records. As there is no auto-removal and it could cause OOME.

Default cache size is 2048 records and there is a DBMaker parameter to regulate its size. This cache is on by default, so it does not need to be enabled in DBMaker. An example:

```
DB db = DBMaker
        .memoryDB()
        .cacheHashTableEnable()
        .cacheSize(1000000)     //optionally change cache size
        .make();
```

5.2 Least-Recently-Used cache

LRU cache keeps track when records are used, and if cache grows beyond maximal size, it removes the least recently used records. Compared to HashTable, cache has better hit statistics, but more overhead. There is overhead associated with maintaining LRU queue. It is recommended to use this cache only if the cost of the deserialization is high and the cache miss would be a serious problem.

MRU queue is maintained on 'best effort' and there are some shortcuts for better performance. So the actual cache size oscillates a few records around maximal size.

This cache is activated by the `cacheLRUEnable()` DBMaker parameter. You can also change the maximal size. The default size is 2048 records. An example:

```
DB db = DBMaker
        .memoryDB()

        .cacheLRUEnable()
        .cacheSize(1000000)     //optionally change cache size

        //optionally enable executor, so cache is cleared in background thread
        .cacheExecutorEnable()

        .make();
```

5.3 Hard reference cache

HardRef cache is an unbounded cache which does not evict any of its entries. This cache is practically a `Map` of recids and records: `HashMap<recid, Record>`

If the store is larger than the heap, it will get filled and eventually cause `OOME` exception. It is great to get great performance from small stores. After the cache is warmed, it offers read-only performance comparable to `java.util` collections.

MapDB also has weak/soft reference caches, but GC has some serious overhead. So hard reference cache has lower overhead if there is enough memory.

This cache is activated with `cacheHardRefEnable()` DBMaker parameter. It does not have maximal size. An example:

```
DB db = DBMaker
        .memoryDB()
        .cacheHardRefEnable()
        //optionally enable executor, so cache is cleared in background thread
```

```
        .cacheExecutorEnable()
        .make();
```

TODO is this cache cleared if free memory is bellow threshold?

No records are automatically removed from this cache. But you can still clear all the records manually: `db.getEngine().clearCache();`

5.4 Soft and Weak reference cache

Another option is to use weak or soft reference cache. Garbage collector removes records from cache after they are GCed. This cache is practically a `Map` of recids and references to records: `HashMap<recid, SoftRef<Record>>`.

Soft and Weak references differ by the eagerness with which they are garbage collected. Weak reference should be GCed immediately after all the references are released. Soft reference should be only removed when the free heap is low. However, the practical implications depend on JVM settings. For example you can still get `OutOfMemoryError` even with soft references, if memory is not reclaimed fast enough.

This caches are activated by `cacheWeakRefEnable()` and `cacheSoftRefEnable()` DBMaker parameters:

```
DB db = DBMaker
        .memoryDB()

        //enable Weak Reference cache
        .cacheWeakRefEnable()
        //or enable Soft Reference cache
        .cacheSoftRefEnable()

         //optionally enable executor, so cache is cleared in background thread
        .cacheExecutorEnable()

        .make();
```

5.5 Disabled cache or reduce size

Instance cache is disabled by default, it was causing memory problems on small devices. On the other hand a disabled cache hurts the performance. So there are could be better alternatives:

First you could clear the cache after every operation. The cache can be cleared using: `db.getEngine().clearCache()`

Another alternative is to reduce the cache size. By default the cache size is 2048 records (when enabled), probably too much for most Android phones:

```
DB db = DBMaker
        .fileDB(file)      //or memory db
        .cacheSize(128)    //change cache size
        .make();
```

5.6 Clear cache

If you use MapDB in batch operations, you can reduce the cache memory overhead, by clearing cache at the end of each batch. If the cache is empty, it gets faster populated by new content from a new batch:

```
//do some heavy stuff with mapdb:
map.getAll...

//we are done clear cache
db.getEngine().clearCache();

//now do some other stuff, which does not use MapDB:
save_my_files...
```

5.7 Cache hit and miss metrics

There is option to enable metrics in DBMaker. MapDB will log metrics periodically at info level if enabled.

TODO cache hit/miss statistics.

5.8 Cache priority

TODO Right now there is no record priority for cache.

However it is very easy to add this into MapDB. For example you could give more preferential treatment to btree directory nodes, while leaf nodes would not be cached. All it takes is a few `record instanceof BTreeDirNode` in single class.

TODO MapDB could also have flexible multi level cache layering.

CHAPTER

SIX

BTREEMAP

`BTreeMap` provides `TreeMap` and `TreeSet` for MapDB. It is based on lock-free concurrent B-Linked-Tree. It offers great performance for small keys and has good vertical scalability.

TODO explain compressions

TODO describe B-Linked-Tree

6.1 Parameters

BTreeMap has optional parameters which can be specified by using maker.:

The most important among them are **serializers**. General serialization has some guessing and overhead, so better performance is always achieved if more specific serializers are used. To specify the key and value serializer, use the code bellow. There are dozens ready to use serializers available as static fields on `Serializer` interface:

```
BTreeMap<Long, String> map = db.treeMapCreate("map")
        .keySerializer(Serializer.LONG)
        .valueSerializer(Serializer.STRING)
        .makeOrGet();
```

Another usefull parameter is **size counter**. By default, a BTreeMap does not keep track of its size and calling `map.size()` requires linear scan to count all entries. If you enable size counter, in that case `map.size()` is instant, but there is some overhead on inserts.

```
BTreeMap<Long, String> map = db.treeMapCreate("map")
        .counterEnable()
        .makeOrGet();
```

BTrees store all their keys and values as part of a btree node. The node size affects performance a lot. A large node means that many keys have to be deserialized on lookup. A smaller node loads faster, but makes large BTrees deeper and requires more operations. The default maximal node size is 32 entries and it can be changed in this way:

```
BTreeMap<Long, String> map = db.treeMapCreate("map")
        .nodeSize(64)
        .makeOrGet();
```

Values are also stored as part of BTree leaf nodes. Large values means huge overhead and on single `map.get("key")` 32 values are deserialized, but only a single value returned. In this case, it is better to store the values outside the leaf node, in a separate record. In this case leaf node only has a 6 byte recid pointing to the value.

Large values can also possible be compressed to save space. This example stores places node outside BTree leaf nodes and applies compression on each value:

```
BTreeMap<Long, String> map = db.treeMapCreate("map")
        .valuesOutsideNodesEnable()
        .valueSerializer(new Serializer.CompressionWrapper(Serializer.STRING))
        .makeOrGet();
```

BTreeMap needs to sort its key somehow. By default it relies on `Comparable` interface implemented by most Java classes. In case this interface is not implemented, a key serializer must be provided. One can for example compare Object arrays:

```
BTreeMap<Object[], Long> map = db.treeMapCreate("map")
        // use array serializer for unknown objects
        .keySerializer(new Serializer.Array(db.getDefaultSerializer()))
        // or use serializer for specific objects such as String
        .keySerializer(new Serializer.Array(Serializer.STRING))
        .makeOrGet();
```

Also primitive arrays can be used as keys. One can replace `String` by `byte[]`,which directly leads to better performance:

```
BTreeMap<byte[], Long> map = db.treeMapCreate("map")
        .keySerializer(Serializer.BYTE_ARRAY)
        .makeOrGet();
```

6.2 Key serializers

BTreeMap owns its performance to the way it handles keys. Lets illustrate this on an example with `Long` keys.

A long key occupies 8 bytes after serialization. To minimize space usage one could pack this value to make it smaller. So the number 10 will occupy a single byte, 300 will take 2 bytes, 10000 three bytes etc. To make keys even more packable, we need to store them in even smaller values. The keys are sorted, so lets use delta compression. This will store the first value in full form and then only the differences between consecutive numbers.

Another improvement is to make the deserialization faster. In normal `TreeMap` keys are stored in wrapped form, such as `Long[]`. That has a huge overhead, as each key requires new pointer, class header... BTreeMap will store keys in primitive array `long[]`. And finally if keys are small enough it can even fit into `int[]`. And because an array has better memory locality, there is a huge performance increase on binary searches.

It is simple to do such optimisation for numbers. But BTreeMap also applies that on other keys, such as `String` (common prefix compression,single `byte[]` with offsets), `byte[]`, `UUID`, `Date` etc.

This sort of optimization is used automatically. All you have to do is provide the specialized key serializer: `.keySerializer(Serializer.LONG)`.

There are several options and implementations to pack keys. Have a look at BTreeKeySerializer for more details.

TODO this is major feature, document details and add benchmarks

6.3 Data Pump

TODO data pump

6.4 Fragmentation

A trade-off for lock-free design is fragmentation after deletion. The B-Linked-Tree does not delete btree nodes after entry removal, once they become empty. If you fill a BTreeMap and then remove all entries, about 40% of space will not be released. Any value updates (keys are kept) are not affected by this fragmentation.

This fragmentation is different from storage fragmentation, so `DB.compact()` will not help. A solution is to move all content into a new `BTreeMap`. As it is very fast with Data Pump streaming, new Map will have zero fragmentation and better node locality (in theory disk cache friendly).

TODO provide utils to move BTreeMap content TODO provide statistics to calculate BTreeMap fragmentation

In the future, we will provide BTreeMap wrapper, which will do this compaction automatically. It will use three collections: the first `BTreeMap` will be read-only and will also contain the data. The econd small map will contain updates. Periodically a third map will be produced as a merge of first two, and will be swapped with the primary. `SSTable`'s in Cassandra and other databases works in a similar way.

TODO provide wrapper to compact/merge BTreeMap content automatically.

6.5 Composite keys and multimaps

MapDB 1.0 had tuples that were replaced in 2.0 with `Object[]`.

TODO composite keys

TODO multimap

6.6 Compared to HTreeMap

BTreeMap is better for smaller keys, such as numbers and short strings.

TODO compare to HTreeMap

CHAPTER

SEVEN

HTREEMAP

HTreeMap provides `HashMap` and `HashSet`. It has a great performance with large keys. It also offers entry expiration based on the size or time-to-live

HTreeMap is a *segmented Hash Tree*. Most hash collections use hash table based in fixed size array and when it becomes full, all data has to be moved and rehashed into a new, bigger table. HTreeMap uses auto-expandind Hash Treestructure, so it never needs resizing. It also occupies less space, since empty hash slots do not consume any space. On the other hand, the tree structure requires more seeks and is slower on access. Its performance degrades with size, but the maximal dir node size is 128, so degradation is very small. TODO performance degradation depending on size. Probably log N.

To achieve parallel scalability, HTreeMap is split into 16 segments, each with a separate read-write lock. `ConcurrentHashMap` in JDK 7 works in a similar way. The number of segments (also called concurrency scale) is hard wired into design and cannot be changed. Because all segments share an underlying storage, concurrent scalability is not perfect. Another option is to create each segment with separate storage.

HTreeMap optionally supports entry expiration based on four criteria: maximal map size, time-to-live since last modification and time-to-live since last access. Expired entries are automatically removed. This feature uses FIFO queue and each segment has independent expiration queue. Priority per entry cannot be set.

7.1 Parameters

HTreeMap has a number of parameters to tune its performance. The number of segments (aka concurrency factor) is hard-coded to 16 and cannot be changed. Other params can be set only when the map is created and cannot be changed latter.

The most important are **serializers**. General serialization has some guessing and overhead, so it always has better performance if more specific serializers are used. To specify a key and value serializer, use the code bellow. There are dozens ready to use serializers available as static fields on `Serializer` interface:

```
HTreeMap<String, Long> map = db.hashMapCreate("map")
        .keySerializer(Serializer.STRING)
        .valueSerializer(Serializer.LONG)
        .makeOrGet();
```

HTreeMap is recommended for handling large key/values. In same cases you may want to use compression. Enabling compression store-wide is not always the best option, since a constantly (de)compressing index tree has overhead. Instead, it is better to apply compression just to a specific serializer on key or value. This is done by using serializer wrapper:

```
HTreeMap<Long, String> map = db.hashMapCreate("map")
        .valueSerializer(new Serializer.CompressionWrapper(Serializer.STRING))
        .makeOrGet();
```

Most hash maps uses 32bit hash generated by `Object.hashCode()` and check equality with `Object.equals(other)`. However many classes do not implement those functions correctly, and inconsistent hashing is very bad for persistence, as it could cause data loss. By default, configuration HTreeMap uses a generic key serializer and relies on those methods as well. However, it throws an `IllegalArgumentException` if inconsistent hashing is detected (for example `byte[]` is used without serializer).

If the specialized Key Serializer is defined, HTreeMap relies on it to provide hash code and equality check for keys. For example `Serializer.BYTE_ARRAY` uses `java.util.Arrays.hashCode(byte[])` to generate hash code. This way you can use primitive arrays directly as a key/value without a wrapper. Bypassing wrappers such as `String`, improves performance:

```
HTreeMap<byte[], Long> map = db.hashMapCreate("map")
        .keySerializer(Serializer.BYTE_ARRAY)
        .makeOrGet();
```

Another parameter is the **size counter**. By default HTreeMap does not keep track of its size and calling `map.size()` requires a linear scan to count all entries. You can enable size counter and in that case `map.size()` is instant, but there is some overhead on inserts.

```
HTreeMap<String, Long> map = db.hashMapCreate("map")
        .counterEnable()
        .makeOrGet();
```

And finally some sugar. There is **value creator**, a function to create a value if the existing value is not found. A newly created value is inserted into the map. This way `map.get(key)` never returns null. This is mainly useful for various generators and caches.

```
HTreeMap<String,Long> map = db.hashMapCreate("map")
        .valueCreator(new Fun.Function1<Long, String>() {
            @Override
            public Long run(String o) {
                return 1111L;
            }
        })
        .makeOrGet();
```

Or a more readable version in Java 8:

```
HTreeMap<String,Long> map = db.hashMapCreate("map")
        .valueCreator((key)-> 1111L)
        .makeOrGet();

// this way map.get() returns 1111L if no value is found
map.get("aa"); // 1111L
map.get("bb"); // 1111L

// map now contains ["aa"->1111L, "bb"->1111L]
```

7.2 Entry expiration parameters

`HTreeMap` offers optional entry expiration if some conditions are met. Entry can expire if:

- The number of entries in a map would exceed the maximal size
- An entry exists in the map longer time than the expiration period is. The expiration period could be since the last modification or since the last read access.
- Disk/memory space consumed by Map is bigger then some limit in GB.

There is a shortcut in `DBMaker` to quickly use `HTreeMap` as an off-heap cache with memory size limit:

```
// Off-heap map with max size 16GB
Map cache = DBMaker
        .newCacheDirect(16);
```

This equals to `expireStoreSize` param:

```
HTreeMap cache = db.createHashMap("cache")
        .expireStoreSize(128)
        .makeOrGet();
```

It is also possible to limit the maximal size of a map:

```
HTreeMap cache = db.hashMapCreate("cache")
        .expireMaxSize(128)
        .makeOrGet();
```

And finally you can set an expiration time since the last modification or since the last access.

```
// remove entries 1H after their last modification, or 10 minutes after last get()
HTreeMap cache = db.hashMapCreate("cache")
        .expireAfterAccess(1, TimeUnit.HOURS)
        .expireAfterWrite(10, TimeUnit.MINUTES)
        .makeOrGet();
```

TODO expiration counts are approximate. Map size can go slightly over limits for short period of time.

TODO disk space limit has issues. Investigate how it works and document

TODO expiration threads single and multithreaded.

7.3 Binding and overflow

Maps in MapDB support call back notification on insert, update and removal. There is `Bind` class (TODO link to chapter), which links two collections together and keeps them synchronized. There are several ways to associate collections but this is described in a separate chapter (TODO link).

Specific for `HTreeMap` is the expiration overflow. It is possible tokeep the most recently used entries in memory for faster access. After the entry expires, it is moved to a slower storage on disk.

To create overflow you need two maps. One on-disk and one in-memory. You bind them together with the `.expireOverflow(onDisk, true)` parameter in builder:

```
DB dbDisk = DBMaker
        .fileDB(file)
        .make();

DB dbMemory = DBMaker
        .memoryDB()
        .make();

// Big map populated with data expired from cache
HTreeMap onDisk = dbDisk
        .hashMapCreate("onDisk")
        .make();

// fast in-memory collection with limited size
HTreeMap inMemory = dbMemory
```

```
        .hashMapCreate("inMemory")
        .expireAfterAccess(1, TimeUnit.SECONDS)
        //this registers overflow to `onDisk`
        .expireOverflow(onDisk, true)
        //good idea is to enable background expiration
        .executorEnable()
        .make();
```

Once binding is established, every entry removed from `inMemory` map will be added to `onDisk` map. This applies to expired entries, but also entries removed using the `map.remove()` method. To completely remove entry from both maps, one must first remove from `inMemory` and then from `onDisk`:

```
//first remove from inMemory
inMemory.remove("key");
//key will be moved to onDisk after deletion by modification listener, remove from onDisk
onDisk.remove("key");
```

If the `inMemory.get(key)` is called, one might expect null, and than check `onDisk`. However to make things simpler `inMemory` has a value creator. So `inMemory.get(key)` returns values from both `inMemory` and `onDisk`. If value is not found, it checks the second map and adds its value into `inMemory`.

```
onDisk.put(1,"one");    //onDisk has content, inMemory is empty
inMemory.size();        //> 0
// get method will not find value inMemory, and will get value from onDisk
inMemory.get(1);        //> "one"
// inMemory now caches result, it will latter expire and move to onDisk
inMemory.size();        //> 1
```

Get and removal synchronizes two collections in a cyclic way. The entry gets moved into `onDisk` when its deleted from `inMemory`. And `inMemory` gets populated from `onDisk` when get method does not find a value in-memory. This creates a consistency problem, as to which map is authoritative? Which contains the 'real' data?

MapDB offers three alternatives. The method `.expireOverflow(onDisk, true)` takes a boolean parameter to control how expiration behaves. With `false` expiration it uses `map.put(key,value)` to insert data to inDisk. With `true` it uses `map.putIfAbsent(key,value`, so no existing value gets overwritten.

The first option is `true`. It means that `inMemory` is authoritative and `onDisk` content will get overwritten once the data expires from memory and overflows to disk:

```
HTreeMap inMemory = dbMemory
        .hashMapCreate("inMemory")
        .expireOverflow(onDisk, true) // <<< true here
        .make();

//add two different entries
onDisk.put(1, "uno");
inMemory.put(1, "one");
//simulate expiration by removing entry
inMemory.remove(1);
//data onDisk are overwritten, inMemory wins
onDisk.get(1);      //> "one"
// inMemory gets repopulated from onDisk
inMemory.get(1);    //> "one"
```

With `false`, the data added into `inMemory` are not considered authoritative. OnDisk content will never get overwritten. This also adds a performance bonus on expiration if the values are the same, since on-disk values are not overwritten by equal values.

```
HTreeMap inMemory = dbMemory
        .hashMapCreate("inMemory")
        .expireOverflow(onDisk, false) // <<< false here
        .make();

//add two different entries
onDisk.put(1, "uno");
inMemory.put(1, "one");
//simulate expiration by removing entry
inMemory.remove(1);
//data onDisk are not overwritten, inMemory loses
onDisk.get(1);      //> "uno"
// inMemory gets repopulated from onDisk
inMemory.get(1);    //> "uno"

//add stuff to inMemory and expire it
inMemory.put(2,"two");
inMemory.remove(2);
//onDisk still gets updated, because it did not contained this key
onDisk.get(2); //> two
```

There is a question into which collection new data should be inserted. It depends how much you value your data. If it caches some external source (such as SQL), I would insert data into `inMemory` and let them hoover in air. In this case data might be inconsistent after crash/shutdown. It is better to drop the cache content and repopulate it from the primary store.

If thee content of the cache is valuable (is primary content, or too high cost to repopulate from SQL) one should insert data to `onDisk`. Disk store can be protected by transaction with periodic commit. In this case only a small time interval of data would be lost.

Inserts are simple, but updates create a consistency problem. If the key-value pair changes, one collection might contain an older value. So on updates it is recommended to update BOTH `inMemory` and `onDisk` collections:

```
//put value to on disk
onDisk.put(1, "one");
//in memory gets updated from on disk, no problem here
inMemory.get(1); //> "one"

//updating just one collection creates consistency problem
onDisk.put(1,"uno");
//old content of inMemory has not expired yet
inMemory.get(1); //> "one"

//one has to update both collections at the same time
onDisk.put(1,"uno");
inMemory.put(1,"uno");
```

The third option is to use listeners and `Bind` utilities yourself.

7.4 Concurrent scalability

HTreeMap scales concurrently by using 16 separate segments, each with its own `ReadWriteLock`. Each segment has its own independent state, hash tree and also expiration queue. But all segments still share underlying storage and are limited by its performance.

There is an option to shard HTreeMap. Each separate segment can get its own storage, so no shared state exist between segments. This way one can get linear concurrent scalability which corresponds to 16 segments. TODO benchmarks.

Trade off in this case is a higher memory consumption. There are 16 different stores, each with its own memory allocator and unused blocks. TODO memory benchmarks. But each store can be compacted separately. TODO add compaction doc for this

```
Map<String, byte[]> map = DBMaker
        .hashMapSegmentedMemory()
        .keySerializer(Serializer.STRING)
        .valueSerializer(Serializer.BYTE_ARRAY)
        .make();
```

7.5 Compared to BTreeMap

HTreeMap has a major advantage over *BTreeMap* with large keys. Unlike BTreeMap, it only stores hash codes in tree nodes. BTreeMap deserializes tree nodes, together with their keys, on each lookup. A simple `BTreeMap.get(key)` could deserialize houndreds of keys.

TODO link to performance test, compare with BTreeMap

On the other side HTreeMap has a limited concurrency factor to 16, so with writes it wont scale over 4 CPU cores. It uses read-write locks, so read operations are not affected. However, in practice the disk IO is more likely to be bottleneck. TODO benchmarks

HTreeMap can be easily sharded by segments. For in-memory map it might have better concurrent scalability.

HTreeMap is simpler than BTreeMap. It has more predictable performance over a long period of time and it does not get fragmented after frequent deletes. HTreeMap also offers expiration. BTreeMap pays tax in some cases for its complex lock-free design.

CHAPTER

EIGHT

SECONDARY COLLECTIONS

Rational databases have a very good system of primary and secondary indexes, tables and views. It has clear benefits for extensibility, clarity and robustness. On the other hand, it has limitations for scalability and performance. MapDBs Secondary Collections are *poor man's* SQL tables. It brings the most benefits without sacrificing flexibility.

Secondary Collections are a very simple and flexible way to access and expand primary collections. Primary collection is an authoritative source of data, and is modified by the user. Secondary collection contains records derived from the primary. Secondary is bind to primary and updated by a listener, when the primary collection is modified. The secondary should not be modified directly by a user.

Secondary Collections are typically used in three ways:

- Secondary Keys (indexes) to efficiently access records in primary collection.
- Secondary Values to expand primary collection, while keeping primary small and fast.
- Aggregation to groups. For example in order to list all high-income customers.

The Primary Collection is typically stored in MapDB. The requirement is that it provides modification listener triggered on entry update, insert or removal. There is no such requirement for the Secondary Collection. The Secondary may be stored in MapDB, but it can also be a usual Java Collection such as java.util.TreeSet.

Primary and Secondary collections are bind together. There is Bind class with static methods to establish binding. Binding adds a modification listener to the primary collection and changes secondary collection accordingly. It also removes old entries from the secondary, if the primary entry gets deleted or modified.

Bind relation is not persistent, so binding needs to be restored every time the store is reopened. If the secondary collections is empty when binded, then the entire primary is traversed and the secondary is filled accordingly.

8.1 Consistency

The consistency between primary and secondary collections are on a 'best-effort' basis. The secondary map might contain an old value, while the primary map was already updated. Also if two maps are part of two different transactions, and one transaction get rolled back but second transaction is commited, the secondary map will become inconsistent with the primary (its changes were not rolled back.

The secondary collection is updated in serializable fashion. This means that if two concurrent threads update primary, the secondary collection is updated with 'winner'. TODO verify this is true, update this paragraph after issue is closed.

There are some best practices for Secondary Collections to handle this:

- Secondary Collections must be thread safe. Either use MapDB or `java.util.concurrent.*` collections. Another option is to use `Collections.synchronized*()` wrappers.
- When using concurrent transactions, do not mix the collections from multiple transactions. If the primary gets rollback, the secondary will not be updated if its not within the same transaction.

- Keep binding minimal. It should only transform one value into the other, without dependency on third collections.

8.2 Performance

To import a large dataset, you should not enable binding until the primary collection has finished its import. Also, there might be a more efficient way to pre-fill the secondary collection (for example with a data pump).

CHAPTER

NINE

TRANSACTIONS

Transactions in MapDB are considered a heavyweight option. Often you can get a similar result using atomic updates in ConcurrentMap and Atomic variables

On the other hand transactions are not just for concurrent updates, but they can also protect the store from corruption. MapDB has two ways to handle rollback: Write-Ahead-Log and Append-Only-Log files.

MapDB comes with several transactions options. From ultra-fast direct mode to concurrent transactions with MVCC snapshots and serializable conflict resolution.

There are three transactional modes:

9.1 Transactions disables (aka direct mode)

In this mode transactions and all safety options are disabled to achieve best write performance. Changes are written directly to underlying files.

This mode has zero crash protection. If you do not close your database correctly, your data are gone. You may also call `db.commit()` to flush and sync all changes into underlying files.

This mode is generally recommended for cases where data can be reconstructed easily. For example initial imports, caches etc.

An example how to enable this mode:

```
DB db = DBMaker.newMemoryDB()
    .transactionsDisable()
    .make();
```

9.2 Single global transaction

This mode is the default option. In this case DB has a single global transaction. It is very simple to use, since the user does not have to handle concurrent write conflicts and rollback. In this case the store is protected by Write-Ahead-Log.

```
DB db = DBMaker.newMemoryDB()
    .make()
```

9.3 Concurrent transactions

In this case the user has multiple transactions. Each has its own snapshots with serializable isolation and optimistic locking. It offers the strongest consistency possible. Any changes are held in memory. On commit MapDB checks for

conflicts and commits or throws `TxRollbackException` and rollback changes.

```
    //Open Transaction Factory. DBMaker shares most options with single-transaction mode.
        TxMaker txMaker = DBMaker
                .newMemoryDB()
                .makeTxMaker();

        // Now open first transaction and get map from first transaction
        DB tx1 = txMaker.makeTx();

        //create map from first transactions and fill it with data
        Map map1 = tx1.getTreeMap("testMap");
        for(int i=0;i<1e4;i++){
            map1.put(i,"aaa"+i);
        }

        //commit first transaction
        tx1.commit();

        // !! IMPORTANT !!
        // !! DB transaction can be used only once,
        // !! it throws an 'already closed' exception after it was commited/rolledback
        // !! IMPORTANT !!
        //map1.put(1111,"dqdqwd"); // this will fail

        //open second transaction
        DB tx2 = txMaker.makeTx();
        Map map2 = tx2.getTreeMap("testMap");

        //open third transaction
        DB tx3 = txMaker.makeTx();
        Map map3 = tx3.getTreeMap("testMap");
//TODO more from tx example
```

CHAPTER

TEN

MAPDB INTERNALS

This chapter gives a quick introduction to MapDB internal architecture. The rest of this manual assumes that you are familiar with this chapter.

MapDB originally evolved as a store for astronomical desktop applications. Over time it grew into a full database engine with concurrent access, durability etc. But it evolved differently from most DBs, Pentium at 200MHz with 128 MB RAM does not give much space. The major goal was tight integration with Java and to minimize overhead of any sort (garbage collection, memory, CPU, stack trace length...).

What makes MapDB most different is that the serialization lifecycle is very different here. In most DB engines the user has to serialize data himself and pass binary data into db. API that looks similar to this:

```
engine.update(long recid, byte[] data);
```

But MapDB serializes data itself by using a user supplied serializer:

```
engine.update(long recid, Person data, Serializer<Person> serializer);
```

So serialization lifecycle is driven by MapDB rather than by the user. This small detail is the reason why MapDB is so flexible. For example the `update` method could pass the data-serializer pair to background-writer thread and return almost instantly. Or `Person` instance could be stored in instance cache, to minimise deserilization overhead on multiple reads. `Person` does not even have to be serialized, but could be stored in `Map<Long,Person>` map on heap, in this case MapDB has speed comparable to Java Collections.

Colons can be used to align columns.

10.1 Dictionary

Term	Explanation
Record	Atomically stored value. Usually tree node or similar. Transaction conflicts and locking is usually per record.
Index Table	Table which translates recid into real offset and size in physical file.
Engine	Primitive key-value store used by collections for storage.
Store	Engine implementation which actually persist data. Is wrapped by other Engines.
Volume	Abstraction over ByteBuffer or other raw data store. Used for files, memory, partition etc..
Slice	Non overlapping pages used in Volume. Slice size is 1MB. Older name was 'chunk'
Direct mode	With disabled transactions, data are written directly into file. It is fast, but store is not protected from corruption during crashes.
WAL	Write Ahead Log, way to protect store from corruption if db crashes during write.
RAF	Random Access File, way to access data on disk. Safer but slower method.
MMap file	memory mapped file. On 32bit platform has size limit around 2GB. Faster than RAF.
index file	contains mapping between recid (index file offset) and record size and offset in physical file (index value). Is organized as sequence of 8-byte longs
index value	single 8-byte number from index file. Usually contains record offset in physical file.
recid	record identificator, a unique 8-byte long number which identifies record. Recid is offset in index file. After record is deleted, its recid may be reused for newly inserted record.
record	atomical value stored in storage (Engine) identified by record identifier (recid). In collections Record corresponds to tree nodes. In Maps record may not correspond to Key->Value pair, as multiple keys may be stored inside single node.
physical file	Contains record binary data
cache (or instance cache)	caches object instances (created with 'new' keyword). MapDB does not have traditional fixes-size-buffer cache for binary pages (it relies on OS to do this). Instead deserialized objects are cached on heap to minimise deserialization overhead. Instance cache is main reason why your keys/values must be immutable.
BTreeMap	tree implementation behind TreeMap and TreeSet provided by MapDB
HTreeMap	tree implementation behind HashMap and HashSet provided by MapDB
delta packing	compression method to minimalise space used by keys in BTreeMap. Keys are sorted, so only difference between keys needs to be stored. You need to provide specialized serializer to enable delta packing.
append file db	alternative storage format. In this case no existing data are modified, but all changes are appended to end of file. This may improve write speed and durability, but introduces some tradeoffs.
temp map/set...	collection backed by file in temporary directory. Is usually configured to delete file after close or on JVM exit. Data written into temp collection are not persisted between JVM restarts.
async write	writes may be queued and written into file on background thread. This does not affect commit durability (it blocks until queue is empty).
TX	equals to Concurrent Transaction.
LongMap	specialized map which uses primitive long for keys. It minimises boxing overhead.
DB	API class exposed by MapDB. It is an abstraction over Engine which manages MapDB collections and storage.
DMaker	Builder style factory class, which opens and configures DB instances.
Collection Binding	MapDB mechanism to keep two collections synchronized. It provides secondary keys and values, aggregations etc.. known from SQL and other databases. All functions are provided as static methods in Bind class.
Data Pump	Tool to import and manipulate large collections and storages.

10.2 DBMaker and DB

90% of users will only need two classes from MapDB. DBMaker is a builder style configurator which opens the database. DB represents the store, it creates and opens collections, commits or rollbacks data.

MapDB collections use `Engine` (simple key-value store) to persist its data and state. Most of the functionality comes from mixing `Engine` implementations and wrappers. For example, off-heap store with asynchronous writes and instance cache could be instantiated by this pseudo-code:

```
//TODO this is obsolete, cache and async are integrated to Store

Engine engine = new Caches.HashTable(         //instance cache
                    new AsyncWriteEngine(     //asynchronous writes
                     new StoreWAL(            //actual store with WAL transactions
                      new Volume.MemoryVol()  //raw buffer used for storage
                )))
```

Reality is even more complex, since each wrapper takes extra parameters and there are more levels. So `DBMaker` is a factory which takes settings and wires all MapDB classes together.

`DB` has a similar role. It is too hard to load and instantiate collections manually (for example `HTreeMap` constructor takes 14 parameters). So `DB` stores all the settings in the Named Catalog and handles collections. Named Catalog is a `Map<String,Object>` which is persisted in store at fixed recid and contains parameters for all other named collections and named records. In order to rename a collection one just has to rename the relevant keys in the Named Catalog.

10.3 Layers

MapDB stack is a little bit different from most DBs. It integrates instance cache and serialization usually found in ORM frameworks. On the other hand MapDB eliminated fixed-size page and disk cache.

From raw-files to `Map` interface it has the following layers:

1. **Volume** - an `ByteBuffer` like abstraction over raw store. There are implementations for in-memory buffers or files.
2. **Store** - primitive key-value store (implementation of `Engine`). Key is offset on index table, value is variable length data. It has single transaction. Implementations are Direct, WAL, append-only and Heap (which does not use serialization). It performs serialization, encryption and compression.
3. **AsyncWriterEngine** - is optional `Store` (or `Engine`) wrapper which performs all modifications on background thread.
4. **Instance Cache** - is `Engine` wrapper which caches object instances. This minimises deserilization overhead.
5. **TxMaker** - is `Engine` factory which creates fake `Engine` for each transaction or snapshot. Dirty data are stored on heap.
6. **Collections** - such as TreeMap use `Engine` to store their data and state.

10.4 Volume

`ByteBuffer` is the best raw buffer abstraction Java has. However, its size is limited by 31 bits addressing to 2GB. For that purpose MapDB uses `Volume` as raw buffer abstraction. It takes multiple `ByteBuffers` and uses them together with 64bit addressing. Each `ByteBuffer` has 1GB size and represents a *slice*. The IO operations which

cross slice boundaries are not supported (`readLong(1GB-3)` will throw an exception at such offset). It is the responsibility of the higher layer `Store` to ensure data do not overlap slice boundaries.

MapDB provides some Volume implementations: heap buffers, direct (off-heap) buffers, memory mapped files and random access file. Each implementation fits a different situation. For example memory mapped files have great performance, however a 32bit desktop app will probably prefer random access files. All implementations share the same format, so it is possible to copy data (and entire store) between implementations.

Users can also supply their own `Volume` implementations. For example, each 1Gb slice can be stored in a separate file on multiple disks, to create software RAID. `Volume` could also handle duplication, binary snapshots (MapDB snapshots are at different layer) or raw disks.

10.5 Store

Engine (and Store) is a primitive key-value store which maps recid (8-byte long record id) to some data (record). It has 4 methods for CRUD operations and 2 transaction methods:

```
long put(A, Serializer<A>)
A get(long, Serializer<A>)
void update(long, A, Serializer<A>)
void delete(long, Serializer<A>)

void commit()
void rollback()
```

By default MapDB stack supports only a single transaction. However there is the wrapper `TxMaker`, which stores un-commited data on heap and provides concurrent ACID transactions.

DB is a low level implementation of the `Engine`, which stores data on raw `Volume`. It usually has two files (or Volumes): index table and physical file. Recid (record ID) is a usually fixed offset in the index table, which contains a pointer to the physical file.

MapDB has multiple `Store` implementations, which differ in speed and durability guarantees. User can also supply their own implementation.

First (and default) is StoreWAL. In this case the Index Table contains the record size and offset in a physical file. Large records are stored as a linked list. StoreWAL has free space management, so released space is reused. However, over time it may require compaction. StoreWAL stores modifications in *Write Ahead Log*, which is a sequence of simple instructions, such as *write byte at this offset*. On commit (or reopen) WAL is replayed into the main store, and discarded after successful file sync. On rollback the WAL is discarded.

StoreDirect shares the same file format with `StoreWAL`, however it does not use write ahead log. Instead, it writes data directly into files and performs file sync on commit and close. This implementation trades any sort of data protection for speed, so data are usually lost if `StoreDirect` is not closed correctly (or synced after last write). Because there is no WAL, this store does not support rollback. This store is used if transactions are disabled.

The third implementation is `StoreAppend`, which provides append-only file store. Because data are never overwritten, it is very solid and stable. However space usage skyrockets, since it stores all modifications ever made. TODO This store is not finished yet, so for example advanced compaction is missing. TODO Also all possibilities of this store are not explored (and documented yet). This store reads all data in sequence, in order to build Index Table which points to newest version of each record. The Index Table is stored on heap.

10.6 TxMaker

MapDB `Store` support only a single transaction. So concurrent transactions need to be serialized and commited one by one. For this there is TxMaker. It is a factory which creates a fake `Engine` for each transaction. Dirty (uncommited) data are stored on heap. Optimistic concurrency control is used to detect conflicts. TxRollbackException is thrown on write or commit, if the current transaction was rolled back thanks to an conflict.

TxMaker has a Serializable Isolation level and this level supports highest guarantees. Other isolation levels are not implemented, since the author does not want to support (and explain) isolation problems.

TODO Current TxMaker uses global lock, so concurrent performance sucks. It will be rewritten after 1.0 release.

10.7 Collections

MapDB collection uses `Engine` as its parameter. There are two basic indexes:

BTreeMap is an ordered B-Linked-Tree. It offers great concurrent performance. It is best for small sized keys.

HTreeMap is a segmented Hash-Tree. It is good for large keys and values. It also supports entry expiration based on maximal size or time-to-live.

There also also Queues and Atomic variables

TODO explain collections.

10.8 Serialization

MapDB contains its own serialization framework. TODO explain serialization

10.9 Concurrency patterns

TODO concurrency patterns.

CHAPTER

ELEVEN

CONCURRENCY

All classes in MapDB are thread-safe and to some extend vertically scalable. Reads should be linearly scalable without limitations. Writes should scale linearly up to 4 CPU cores, with a peak around 6 cores. Greater write scalability is possible with a sharding or experimental memory allocator.

However MapDB design is relatively simple. There are global locks and stop-word states. It is also flexible so it can be tuned well into most situations and performance / safety compromises.

11.1 Segment locking

Here is a basic explanation of how MapDB can scale under concurrent access. {@javadoc Engine} implements key-value storage. Instead of a single global lock, it is split into multiple segments, each withs its own `ReadWriteLock`

This way, the records can be updated concurrently to some extend. Concurrent read operations are not blocked unless an update is running.

Under the segment locks is a global lock called 'structural lock' which protects the memory allocator. It is locked when record layout changes, for example when space is allocated, a record was resized or free space is released.

```
// read record from store
locks[recid % locks.length].readLock().lock(); //note readLock
try{
    //look up recid, deserialize and return
}finally {
    locks[recid % locks.length].readLock().unlock();
}

// update record from store
locks[recid % locks.length].writeLock().lock();
try{

    //TODO finish update example
}finally {
    locks[recid % locks.length].readLock().unlock();
}
```

The memory allocator uses single global lock, and that limits vertical scalability. There are some workarounds. For example, each segment can have its own storage file, so no shared state exists.

11.2 Collection locking

On top of `Engine` collections have their own locking layer. Each collection adds its own locking to ensure that its data (tree nodes, pointers etc) are consistent.

`BTreeMap` (aka TreeMap) is a lock free BTree implementation. It only locks a single node on update and has excellent vertical scalability.

`HTreeMap` splits its data into 16 segments based on key hash. The number of segments is hardcoded into its design and can not be changed. However it has a simpler design compared to `BTreeMap` and more predictable performance. It is also easy to use separate storage for each segment in order to top boost its performance.

Please note that locking at collection layer might not use memory barrier. Since collections are stateless (all state is in database), locks at this level only protects from concurrent modification.

11.3 Consistency safety

`Map.put()` translates into multiple operations at `Engine` level. For example a tree must be updated at more levels, counters incremented etc. Under some configurations this might not be an atomic operation and could cause data inconsistency.

For example two commits might happen while a tree is being updated. The second commit is rolled back and the tree becomes inconsistent. Or if an inconsistent snapshot is taken, while the tree is being updated, that would cause an inconsistent backup, while the primary data could be fine.

There are two ways to solve this. First are the sequential safe collections (usually lock-free), which can handle inconsistency at the expense of fragmentation and lower performance. `BTreeMap` is sequentially safe, its nodes are updated in a way which handles inconsistency.

The second way is *Consistency Lock*. It is `ReadWriteLock` provided by `DB.getConsistencyLock()`. Sequentially, unsafe operations (such as `HTreeMap.put()`) should be performed under read lock. Operations which require a consistent state (taking snapshot, commit or close) should be performed under write lock.

`DB` provides consistency lock:

```
DB db = DBMaker.memoryDB().make();

// there are two counters which needs to be incremented at the same time.
Atomic.Long a = db.atomicLong("a");
Atomic.Long b = db.atomicLong("b");


// update those two counters together
db.consistencyLock().readLock().lock(); //note readLock
try{
    a.incrementAndGet();
    // if snapshot or commit would happen here, two counters would be inconsistent
    b.incrementAndGet();
}finally {
    db.consistencyLock().readLock().unlock();
}

//now backup two counters (simulates taking snapshot)
db.consistencyLock().readLock().lock();  //not writeLock
try{
    System.out.println(
            a.get() + " = " + b.get()
```

```
    );
}finally {
    db.consistencyLock().readLock().unlock();
}
```

Please note that Consistency Lock might not use memory barrier. Since collections are stateless (all state is in database), locks at this level only protects from concurrent modification. For details consult MapDB source code.

11.4 Executors

Some tasks, such as write or queue maintenance, could be done in a background thread. In default configuration MapDB does not start any threads (its safer), but piggy backs those tasks as part of regular operations. This might slow down MapDB operations.

Another option is to give MapDB an executor which performs tasks on the background. In this case some tasks will be executed asynchronously (writes) or in parallel (compaction). The simplest way is to enable the executor globally:

```
DB db = DBMaker
        .memoryDB()
        //enable executors globally
        .executorEnable()
        .make();
```

Concurrent execution usually improves the performance, but sometimes it might make it worse. For that, MapDB has several options to enable Executor only for specific tasks. For example a parallel compaction has no benefit for in-memory store, but on-disk it brings large improvement, since IO (and disk seeks) can be done in parallel:

```
DB db = DBMaker
        .memoryDB()

        //enable executor used for compaction
        .storeExecutorEnable()
        //or use your own executor
        .storeExecutorEnable(
                Executors.newSingleThreadScheduledExecutor()
        )
        .make();
//perform compaction
db.compact();
```

Another option is to enable executor just for asynchronous writes:

```
DB db = DBMaker.memoryDB()
        //TODO specific executor for async write


        .make();
```

And finally, for cache expiration in reference based cache (hard, soft or weak cache):

```
DB db = DBMaker.memoryDB()
        // enable executor just for instance cache
        .cacheExecutorEnable()
        // or one can use its own executor
        .cacheExecutorEnable(Executors.newSingleThreadScheduledExecutor())

        //only some caches are using executor for its expirations:
```

```
        .cacheHardRefEnable()    //TODO check hardref cache uses executors
        .cacheLRUEnable()        //TODO check LRU cache uses executors
        .cacheWeakRefEnable()
        .cacheSoftRefEnable()

        .make();
```

There is also the question about how many threads and what type of Executor one should use. MapDB uses ScheduledExecutorService to execute its tasks. Previous MapDB versions were starting threads directly, but that meant problems in restricted and managed environments such as J2EE containers. It is possible to supply your own Executor directly, as usually you can use one of the factory methods in `Executors`:

```
DB db = DBMaker
        .memoryDB()
        //this would just enable global executor with default value
        // .executorEnable()
        //this will enable global executor supplied by user
        .executorEnable(
                //TODO Executors.newSingleThreadScheduledExecutor()
        )
        .make();

//remember that executor gets closed on shutdown
db.close();
```

TODO how many threads and executor selection

The Executor keeps threads running on the background. That could mean memory leak and also prevents JVM from shutting down. And if executor keeps running after db is closed, background tasks would fail with an exception and possibly loose unwritten data. For that reason `DB.close()` shutdowns all Executors associated with DB instance, before it is closed. This way all background task are finished (and synced) before DB is closed.

Currently it is not possible to reuse Executor after DB was closed. It is also not possible to share Executor between DB instances if one of them is closed. The solution for that is to provide Executor wrapper, which only shutdowns its own tasks, but that is not implemented yet.

TODO executor sharing wrapper

CHAPTER

TWELVE

APPENDIX: STORAGE FORMATS

12.1 Parity

MapDB uses parity bits to check storage pointers for corruption. Offsets are usually aligned to multiples of 8, so lower bits are used for parity checks and other flags.

One important requirement is that zeroes is not valid parity value (0 produces non zero parity bits).

12.1.1 Parity 1

Used for values where last bit is unused, lowest bit stores parity information. It is calculated as number of bits set, lowest bit is than set so total number of non-zero bits is odd:

```
val | ((Long.bitCount(val)+1)%2)
```

Methods: `DataIO.parity1set()` and `DataIO.parity1get()`

TODO document other parity methods, for now see DataIO class

12.2 Feature bitmap header

Feature bitmap is 64bits stored in header at offset 8 (after file header and version). It indicates features storage was created with. Some of those affect storage format (compression, checksums) and must be enabled to make store readable. Some slots are not yet used and are reserved for future features. If such unknown bit is set, MapDB might refuse to open storage, with exception that never version should be used

Currently used feature bits are:

1. LZW record compresson enabled
2. XTEA record encryption enabled. User must supply password to open database.
3. CRC32 record checksum enabled
4. Store does not track free space. There might be unclaimed free space between records, this makes free space metrics invalid.
5. Sharded engine. It means that name catalog or class catalog might not be present
6. Created from backup. Storage was not created empty, but from existing database, as backup or from data pump
7. External index table. Index table is not stored in this Volume (file), but somewhere outside, most likely in external file
8. Compaction disabled. Some features might prevent compaction, for example StoreAppend in single file.

9) Paranoid. Store was created by patched MapDB it added extra information to catch bugs and data corrupton. This store can not be opened with normal mapdb.

10) Disable parity bit checks. Stor was created by patched MapDB. For extra performance some checksums were disabled. This store can not be opened with normal mapdb.

11. Block encryption enabled

TODO declare range which is backward compatible in read-only mode.

12.3 StoreDirect

StoreDirect uses update in place. It keeps track of free space released by record deletion and reuses it. It has zero protection from crash, all updates are written directly into store. Write operations are very fast, but data corruption is almost guaranteed when JVM crashes. StoreDirect uses checksums as passive protection not to return incorrect data after corruption. Internal data corruption should be detected reasonably fast.

StoreDirect allocates space in 'pages' of size 1MB. Operations such as `readLong`, `readByte[]` must be aligned so they do not cross page boundaries.

12.3.1 Head

Header in StoreDirect format is composed by number of 8 byte longs:

0. **header and head checksum. Checksum is CRC of entire HEAD and is recalculated on** every sync/close. Invalid checksum means that store was not closed correctly, is very likely corrupted and MapDB should fail to open it. See `StoreDirect.headChecksum()`
1. bit field indicating **format features**. IE what type of checksums are enabled, compression enabled etc...
2. **store size** pointer to last allocated page inside store. Parity 16.
3. **max recid** maximal allocated recid. Shifted <<<3 for parity 3
4. **index page registry** points to page with list of index pages. Parity 16.
5. **free recids** longs stack

TODO free longstacks

TODO rest of zero page is filled by recids

12.3.2 Index page

Linked list of pages. It starts at **index page registry**.

Index page contains at start:

- first value is **pointer to next index page**, Parity 16
- **second value in page is checksum of all values on page (add all values)** TODO checksum is different

Rest of the index page is filled with index values.

12.3.3 Index Value

Index value translates Record ID (recid) into offset in file and record size. Position and size of record might change as data are updated, that makes index tables necessary. Index Value is 8 byte long with parity 1

- **bite 49-64** - 16 bite record size. Use `val>>48` to get it
- **bite 5-48** - 48 bite offset, records are aligned to 16 bytes, so last four bites can be used for something else. Use `val&MOFFSET` to get it
- **bite 4** - linked or null, indicates if record is linked (see section TODO link to section). Also `linked && size==0` indicates null record. Use `val&MLINKED`.
- **bite 3** - indicates unused (preallocated or deleted) record. This record is destroyed by compaction. Use `val&MUNUSED`
- **bite 2** - archive flag. Set by every modification, cleared by incremental backup. Use `val&MARCHIVE`
- **bite 1** - parity bit

12.3.4 Linked records

Maximal record size is 64KB (16bits). To store larger records StoreDirect links multiple records into single one. Linked records starts with Index Value where Linked Record is indicates by a bit. If this bit is not set, entire record is reserved for record data. If Linked bit is set, than first 8 bytes store Record Link with offset and size of the next part.

Structure of Record Link is similar to Index Value. Except parity is 3.

- **bite 49-64** - 16 bite record size of next link. Use `val>>48` to get it
- **bite 5-48** - 48 bite offset of next record alligned to 16 bytes. Use `val&MOFFSET` to get it
- **bite 4 - true if next record is linked, false if next record is last and not linked (is tail of linked record).** Use `val&MLINKED`
- **bite 1-3** - parity bits

Tail of linked record (last part) does not have 8-byte Record Link at beginning.

12.3.5 Long Stack

Long Stack is linked queue of longs stored as part of storage. It supports two operations: put and take, longs are returned in FIFO order. StoreDirect uses this structure to keep track of free space. Space allocation involves taking long from stack. There are more stacks, each size has its own stack, there is also stack to keep track of free recids. For space usage there are in total `64K / 16 = 4096` Long Stacks (maximal non-linked record size is 64K and records are aligned to 16 bytes).

Long stack is organized similar way as linked record. It is stored in chunks, each chunks contains multiple long values and link to next chunk. Chunks size varies. Long values are stored in bidirectional-packed form, to make unpacking possible in both directions. Single value occupies from 2 bytes to 9 bytes. TODO explain bidi-packing, for now see DataIO class.

Each Long Stack is identified by master pointer, which points to its last chunk. Master Pointer for each Long Stack is stored in head of storage file at its reserved offset (zero page). Head chunk is not linked directly, one has to fully traverse Long Stack to get to head.

Structure of Long Stack Chunk is as follow:

- **byte 1-2** total size of this chunk.
- **byte 3-8** pointer to previous chunk in this long stack. Parity 4, parity is shared with total size at byte 1-2.

- rest of chunk is filled with bidi-packed longs with parity 1

Master Link structure:

- **byte 1-2** tail pointer, points where long values are ending at current chunk. Its value changes on every take/put.
- **byte 3-8** chunk offset, parity 4.

Adding value to Long Stack goes as follow:

1. check if there is space in current chunk, if not allocate new one and update master pointer
2. write packed value at end of current chunk
3. update tail pointer in Master Link

Taking value:

1. check if stack is not empty, return zero if true
2. read value from tail and zero out its bits
3. update tail pointer in Master Link
4. if tail pointer is 0 (empty), delete current chunk and update master pointer to previous page

12.4 Write Ahead Log

WAL protects storage from data corruption if transactions are enabled. Technically it is sequence of instructions written to append-only file. Each instruction says something like: 'write this data at this offset'. TODO explain WAL.

WAL is stored in sequence of files.

12.4.1 WAL lifecycle

- open (or create) WAL
- replay if unwritten data exists (described in separate section)
- start new file
- write instructions as they come
- on commit start new file
- sync old file. Once sync is done, exit commit (it is blocking operation, until data are safe)
- once log is full, replay all files
- discard logs and start over

12.4.2 WAL file format

- **byte 1-4** header and file number
- **byte 5-8** CRC32 checksum of entire log file. TODO perhaps Adler32?
- **byte 9-16** Log Seal, written as last data just before sync.
- rest of file are instructions
- **end of file** - End Of File instruction

12.4.3 WAL Instructions

Each instruction starts with single byte header. First 3 bits indicate type of instruction. Last 5 bits contain checksum to verify instruction.

Type of instructions:

0. **end of file**. Last instruction of file. Checksum is `bit parity from offset & 31`
1. **write long**. Is followed by 8 bytes value and 6 byte offset. Checksum is `(bit parity from 15 bytes + 1)&31`
2. **write byte[]. Is followed by 2 bytes size, 6 byte offset and data itself.** Checksum is `(bit parity from 9 bytes + 1 + sum(byte[]))&31`
3. **skip N bytes. Is followed by 3 bytes value, number of bytes to skip .** Used so data do not overlap page size. Checksum is `(bit parity from 3 bytes + 1)&31`
4. **skip single byte**. Skip single byte in WAL. Checksum is `bit parity from offset & 31`
5. **record. Is followed by 6 bytes recid, than 4 bytes record size and an record data.** Is used in Record format. Size==-2 is tombstone, size==-1 is null record TODO checksum for record inst
6. TODO write two bytes.

12.5 Append Only Store

StoreAppend implements Append-Only log files storage. It is sequence of instructions such as 'update record', 'delete record' and so on. Optionally store can be split between multiple files, to support online compaction.

12.5.1 Instructions

Recid and size has parity. If CRC32 is enabled parity is 16 bites, otherwise 1 bite parity. Their value bite-shifts to make space for parity bits at end.

1. record update. Followed by packed recid with parity, packed size with parity and binary data
2. delete record. Places tombstone in index table. Followed by packed recid with parity.
3. record insert. Followed by packed recid with parity, packed size with parity and binary data
4. preallocate record. Followed by packed recid with parity
5. skip N bytes. Followed by packed size with parity.
6. skip single byte
7. EOF current file. Move to next file
8. Current transaction is valid. Start new transaction
9. Current transaction is invalid. Rollback all changes since end of previous transaction. Start new transaction