Contents

Intro

TODO intro text based on intro video

TODO link intro video

What is new

TODO this will be change-log after 1.0.0 is released.

Getting started

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

Best place to checkout various features of MapDB are Examples. There is also screencast which describes most aspects of MapDB.

There is MapDB Cheat Sheet, on just two pages it is quick reminder of MapDB capabilities.

Maven

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

```
<dependency>
    <groupId>org.mapdb</groupId>
    <artifactId>mapdb</artifactId>
    <version>1.0.6</version>
</dependency>
```

There is also repository with daily builds:

```
<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>1.1.0-SNAPSHOT</version>
    </dependency>
</dependencies>
```

Hello World

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

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

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

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.newFileDB(new File("testdb"))
                .closeOnJvmShutdown()
                .encryptionEnable("password")
                .make();

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

What you should know

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

- Transactions (write-ahead-log) can be disabled with DBMaker.transactionDisable(), this will speedup writes. However without transactions 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 defrag on 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 affects data safety, if you use durable commits.

'DBMaker and DB

MapDB is set of loosely coupled components. One could wire classes such as `CacheMRU`, `StoreWAL` and `BTreeMap` manually, byt there are two factory classes

to do it for you: `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) store use one of `DBMaker.newXXX()` static methods. MapDB has more formats and modes, each `newXXX()` uses different: `newMemoryDB()` opens in-memory database backed by `byte[]`, `newAppendFileDB()` opens db which uses append-only log files and so on.

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

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

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

`getXXX()` opens existing collection (or record). If collection with given name does not exist, it is silently created with default settings and returned. An example:

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

`createXXX()` creates new collection (or settings) 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.createAtomicVar("mainPerson", Person.SERIALIZER);
```

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

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

Transactions

`DB` has methods to handle 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 transactions. Examples above use single global transaction, which is sufficient for some usages. MapDB support concurrent transactions as well with full serializable isolation, optimistic locking and MVCC snapshots. In that case we need one extra factory which creates transactions: `TxMaker`. We use `DBMaker` to create it, but instead of `make()` we call `makeTxMaker()`

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

And `TxMaker` is than used to create multiple `DB` objects, each representing single transaction:

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

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

DB tx2 = txMaker.makeTx();
Map map2 = tx1.getTreeMap("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 happend on large maps with sufficent number of BTree nodes.
```

Quick intro to MapDB internals

This chapter gives 5 minutes introduction to MapDB internal architecture. Rest of this manual assume that you have read this chapter.

MapDB originally evolved as store for astronomical desktop application. As such it has a bit different design from most DBs. Major goal was to minimise overhead of any sort (garbage collection, memory, CPU, stack trace length...).

Also serialization lifecycle is very different here. In most DB engines user has to serialize data himself and pass binary data into db. API than looks similar to this:

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

But MapDB serializes data itself by using user supplied serializer:

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

So serialization lifecycle is driven by MapDB rather than by user. This small detail is reason why MapDB is so flexible. For example `update` method could pass 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.

Dictionary

Term	Explanation
Record	Atomically stored value. Usually tree node or similar. Transaction conflicts and lo
Index Table	Table which translates recid into real offset and size in physical file.
Engine	Primitive key-value store used by collections for storage. Most features are Engine
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, parti
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
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 R
index file	contains mapping between recid (index file offset) and record size and offset in ph
index value	single 8-byte number from index file. Usually contains record offset in physical file
recid	record identificator, an unique 8-byte long number which identifies record. Recid i
record	atomical value stored in storage (Engine) identified by record identifier (recid). In
physical file	Contains record binary data
cache (or instance cache)	caches object instances (created with 'new' keyword). MapDB does not have tradi
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 sor
append file db	alternative storage format. In this case no existing data are modified, but all chan

Term	Explanation
temp map/set...	collection backed by file in temporary directory. Is usually configured to delete file
async write	writes may be queued and written into file on background thread. This does not a
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 Ma
DMaker	Builder style factory class, which opens and configures DB instances.
Collection Binding	MapDB mechanism to keep two collections synchronized. It provides secondary ke
Data Pump	Tool to import and manipulate large collections and storages.

DBMaker and DB

90% of users will only need two classes from MapDB. DBMaker is builder style configurator which opens database. DB represents 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 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:

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

Layers

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

From raw-files to `Map` interface it has 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.

Volume

`ByteBuffer` is 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 `ByteBuffer`s and uses them together with 64bit addressing. Each `ByteBuffer` has 1GB size and represents *slice*. IO operations which cross slice boundaries are not supported (`readLong(1GB-3)` will throw an exception). It is responsibility of 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 different situation. For example memory mapped files have great performance, however 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.

User can also supply their own `Volume` implementations. For example each 1Gb slice can be stored in 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.

Store

Engine (and Store) is 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 single transaction. However there is wrapper `TxMaker` which stores un-commited data on heap and provides concurrent ACID transactions.

DB is low level implementation of `Engine` which stores data on raw `Volume`. It usually has two files (or Volumes): index table and physical file. Recid (record ID) is usually fixed offset in index table, which contains pointer to 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 Index Table contains record size and offset in physical file. Large records are stored as 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 sequence of simple instructions such as *write byte at this offset*. On commit (or reopen) WAL is replayed into 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 data 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.

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.

Engine Wrappers

Big part of features in MapDB is implemented as `Engine` wrappers. For example `update` method does not have modify file directly, but it can forward modification into background-writer

Also deserialized records can be stored in instance cache, so it does not have to be deserialized on next read.

TODO expand Engine Wrappers section

TxMaker

MapDB `Stores` support only single transaction. So concurrent transactions needs to be serialized and commited one by one. For this there is TxMaker. It is factory which creates 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 current transaction was rolled back thanks to an conflict.

TxMaker has Serializable Isolation level, this level supports highest guarantees. Other isolation levels are not implemented, since 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.

Collections

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

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

HTreeMap is 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.

Serialization

MapDB contains its own serialization framework. TODO explain serialization

Concurrency patterns

TODO concurrency patterns.

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.

Transactions disabled

If process dies in middle of write, storage files might become inconsistent. For example pointer was updated with new location, where new data were not written yet. For this MapBD storage is protected by write-ahead-log (WAL) which replies 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 store before JVM shutdown, or you loose all your data. You may also use shutdown hook to close database before JVM exits, however this does not protect your data if JVM crashes or is killed:

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

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

Other important usage is for initial data import. Transactions and no-transactions share the same storage format (except WAL) so one can import

data very fast with transactions disabled. Once import is finished, store gets 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 next best thing: Commit tries to flush all write caches and synchronizes 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.

Memory mapped files (mmap)

MapDB was designed from ground to take advantage of mmap files. However mmap files are limited to 2GB by addressing limit 32bit JVM. Mmap have lot of nasty effects on 32bit JVM, so by default we use slower and safer disk access mode called Random-Access-File (RAF).

Mmap files are much faster compared to RAF. Exact speed bonus depends on 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 64bit platform is detected: `DBMaker.mmapFileEnableIfSupported()`.

And finally you can take advantage of mmap files in 32bit platforms by using mmap file only for small but frequently used part of storage: `DBMaker.mmapFileEnableIfSupported()`

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

CRC32 checksums

You may want to protect your data from disk corruption. MapDB optionally supports CRC32 checksums. In this case each record stores extra 4 bytes which contains its CRC32 checksum. If data are somehow modified or corrupted (file system or storage error), next read will fail with an exception. This gives early warning and prevents db from returning 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 affect storage format, so once activate you always have to reopen store with this setting. Also checksum can not be latter activate once 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?

Transactions

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

On other side transactions are not just for concurrent updates, but also means to protect 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:

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 howto enable this mode:

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

Single global transaction

This mode is default option. In this case DB has single global transaction. It is very simple to use since 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()
```

Concurrent transactions

In this case user has multiple transactions. Each has its own snapshots with serializable isolation and optimistic locking. It offers strongest consistency possible. 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
```

Caches

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

Most dbs and old generation of MapDB (JDBM) use fixed size to cache read from disk. MapDB eliminated page layer, so it does not have regular cache. Instead it used memory mapped files and relies on 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, second time it will be fetched from cache.

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

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

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

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

//right
Person person = map.get("John");
person = person.clone();
person.setAge(15);
map.put("John",person);
```

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

Hash Table cache

This cache is 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, but provides good results, so this cache is on by default.

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

Default cache size is 32000 records, there is 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
    .newFileDB(file)          //or memory db
    .cacheSize(1000000)     //optionally change cache size
    .make()
```

Least-Recently-Used cache

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

MRU queue is maintained on 'best effort', there are some shortcuts for better performance. So the actual cache size oscillates a few records around maximal size. Please consult `LongConcurrentLRUMap` source for example.

This cache is activated by `cacheLRUEnable()` DBMaker parameter. You can also change maximal size, default size is 32000 records. An example:

```
DB db = DBMaker
    .newFileDB(file)        //or memory db
    .cacheLRUEnable()
    .cacheSize(1000000)     //optionally change cache size
    .make()
```

Hard reference cache

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

If store is larger than heap, it will get filled and eventually cause `OOME` exception. It is great to get great performance from small stores. After 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
    .newFileDB(file)          //or memory db
    .cacheHardRefEnable()
    .make()
```

No records are automatically removed from this cache. But you can still clear all records manually:

```
db.getEngine().clearCache();
```

Soft and Weak reference cache

Other option is to use weak or soft reference cache. In this cache garbage collector removes records from cache. This cache is practically `Map` of recids and references to records: `HashMap<recid, SoftRef<Record>>`.

Soft and Weak references differs by eagerness with which they are garbage collected. Weak reference should be GC immediately after all references are

released. Soft reference should be only removed when free heap is low. However practical implications depends on JVM settings. For example you can still get `OutOfMemoryError` even with soft references.

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

```
//weak
DB db = DBMaker
    .newFileDB(file)            //or memory db
    .cacheWeakRefEnable()
    .make()

//or soft
DB db = DBMaker
    .newMemoryDB()              //or file db
    .cacheSoftRefEnable()
    .make()
```

Disabled cache or reduce size

Instance cache is enabled by default. On small devices you may want to disable it to reduce memory usage. This is done by `cacheDisable()` parameter:

```
DB db = DBMaker
    .newFileDB(file)          //or memory db
    .cacheDisable()
    .make()
```

Completely disabling cache hurts performance. So there are could be better alternatives:

First you could clear cache after every operation. Checkout howto clearc cache in chapter bellow:

Other alternative is to reduce cache size. By default cache size is 32,000 records, probably too much for most Android phones:

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

Clear cache

If you only use MapDB in batches, you can reduce cache memory overhead, by clearing cache at end of 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...
```

Cache hit and miss statistics

Right now MapDB does not offer hit/miss statistics for any cache. This feature requires a few lines of code and will be added soon.

TODO cache hit/miss statistics.

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.

//TODO better name, perhaps 'user story'?

BTreeMap

HTreeMap

HTreeMap (aka HashMap) is one of `Map`s offered by MapDB. It has great performance with large keys. It also offers entry expiration if time-to-live or maximal size is exceeded.

HTreeMap is *segmented Hash Tree*. Most hash collections use an array for hash table, which requires copying all data when hash table is resized. HTreeMap uses auto-expanding 4 level tree, so it never needs resizing.

To support concurrency HTreeMap is split to 16 independent segments, each with separate read-write lock. `ConcurrentHashMap` works similar way. Number of segments (also called concurrency factor) is hard wired into design and can not be changed.

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, each segment has independent expiration queue. Priority per entry can not be set.

Parameters

HTreeMap has number of parameters to tune its performance. Number of segments (aka concurrency factor) is hard-coded to16 and can not be changed. Other params can be set only when map is created and can not be changed latter.

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

```
HTreeMap<String, Long> map = db.createHashMap("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 best, since constantly (de)compressing index tree has overhead. Instead it is better to apply compression just to specific large key. This is done by using serializer wrapper:

```
HTreeMap<String, Long> map = db.createHashMap("map")
    .keySerializer(new Serializer.CompressionWrapper(Serializer.STRING))
    .makeOrGet()
```

Other useful parameter is **Hasher**. By default HTreeMap uses 32bit hash generated by `hashCode()`. Some classes just return memory pointer, which changes after serialization/deserialization, rendering it useless for persistence. Also it is not always necessary to calculate full hash of large objects (Strings), sometimes it is better to use weaker hash. So HTreeMap allows you to supply custom Hasher which will generate hash code for keys and decide which keys are equal. This way you can use primitive arrays as key without wrapper:

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

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

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

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

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

or more readable version in Java 8:

```
HTreeMap<String,Long> map = db.createHashMap("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]
```

Entry expiration parameters

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

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

There is shortcut in `DBMaker` to quickly use `HTreeMap` as 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 maximal size of map:

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

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

```
// remove all entries 1H after last modification, or 10 minutes after last get()
HTreeMap cache = db.createHashMap("cache")
     .expireAfterWrite(1, TimeUnit.HOURS)
     .expireAfterRead(10, TimeUnit.MINUTES)
     .makeOrGet()
```

Compared to BTreeMap

HTreeMap has one major advantage for using with large keys. Unlike BTreeMap it only stores hash codes in tree nodes. Each lookup on BTreeMap deserializes number of tree nodes together with their keys.

TODO link to performance test, compare with BTreeMap

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

- Queues
- FIFO Queue
- Stack
- CircularQueue

Queues

MapDB has three `Queue` implementations:

- Queue aka First-In-First-Out
- Stack aka Last-In-First-Out
- CircularQueue with limited size

Lock-free queues in MapDB have several limitations, for example it is not possible to count elements without actually removing them. Queues in MapDB usually implement only minimal subset from `BlockingQueue` interface. In future we will introduce `List` which will fully implement `BlockingDequeue` and `List` interfaces, this will use global `ReadWriteLock`.

To instantiate queue use get method:

```
// first-in-first-out queue
BlockingQueue fifo = db.getQueue("fifo");

// last-in-first-out queue (stack)
BlockingQueue lifo = db.getStack("lifo");

// circular queue with limited size
BlockingQueue c = db.getCircularQueue("circular");
```

FIFO Queue

This only takes two extra parameters. Fist you can supply custom serializers used on entries.

Second param decides lock based eviction. Lock-free (false) means better concurrency, but places tomb-stones in place of removed entries, over time the store size grows and will need compaction. With locks (true) the removed entries are deleted under global locks, so there is concurrency penalty.

```
// first-in-first-out queue
BlockingQueue<String> fifo = db.createQueue("fifo", Serializer.STRING, false);
```

Stack

This only takes two extra parameters. Fist you can supply custom serializers used on entries.

Second param decides lock based eviction. Lock-free (false) means better concurrency, but places tomb-stones in place of removed entries, over time the store

size grows and will need compaction. With locks (true) the removed entries are deleted under global locks, so there is concurrency penalty.

```
// last-in-first-out queue
BlockingQueue<String> stack = db.createStack("stack", Serializer.STRING, false);
```

CircularQueue

This only takes two extra parameters. Fist you can supply custom serializers used on entries.

Second parameter is queue size. If number of entries exceeds the queue size, some entries will get overwritten and lost.

```
// circular queue
BlockingQueue<String> circular = db.createCircularQueue("circular", Serializer.STRING, 1
```

Secondary Collections

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

Secondary Collections are very simple and flexible way to access and expand primary collections. Primary collection is authoritative source of data, and is modified by user. Secondary collection contains records derived from primary. Secondary is bind to primary and updated by listener when primary collection is modified. Secondary should not be modified directly by 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 to list all high-income customers.

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 Secondary Collection. Secondary may be stored in MapDB, but it can also be usual Java Collection such as java.util.TreeSet.

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

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

Consistency

Consistency between primary and secondary collections is on 'best-effort' basis. Two concurrent threads might observe secondary contains old values while primary was already updated. Also if secondary is on heap, while primary is in transactional store which gets rolled back, secondary will become inconsistent with primary (its changes were not rolled back.

Secondary collection is updated in serializable fashion. This means that if two concurrent thread update primary, 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. Other (but not optimal) option is to use `Collections.synchronized*()` wrappers.
- When using concurrent transactions, do not mix collections from multiple transactions. If primary gets rollback, secondary will not be updated if its not within the same transaction.
- Keep binding minimal. It should only transform one value into other, without dependency on third collections.

Performance

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