Table of Contents

Introduction

MapDB is an open-source (Apache 2.0 licensed), embedded Java database engine and collection framework. It provides Maps, Sets, Lists, Queues, Bitmaps with range queries, expiration, compression, off-heap storage and streaming. MapDB is probably the fastest Java database, with performance comparable to `java.util` collections. It also provides advanced features such as ACID transactions, snapshots, incremental backups and much more.

This manual is a work-in-progress and it will be completed together with the MapDB 3.0 release. We hope you will find it useful. If you'd like to contribute to MapDB, we would be very happy to accept pull requests

Code examples from this manual are in github repository

Quick Introduction

MapDB is flexible, with many configuration options. But in most cases, it is configured with just a few lines of code.

TODO more resources: cheat sheet, examples, KATA...

Get it

MapDB binaries are hosted in Maven Central repository. Here is dependency fragment for MapDB.

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

`VERSION` is the last version number from Maven Central. You can also find the current version on this image:

maven central 3.0.2 Daily builds are in snapshot repository. The version number for the latest snapshot is here.

```
<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>VERSION</version>
    </dependency>
</dependencies>
```

You can also download MapDB jar files directly from Maven Central. In that case keep on mind that MapDB depends on Eclipse Collections, Guava, Kotlin library and some other libraries. Here is full list of dependencies for each version.

Hello World

Hereafter is a simple example. It opens in-memory HashMap, it uses off-heap store and it is not limited by Garbage Collection:

```
//import org.mapdb.*
DB db = DBMaker.memoryDB().make();
ConcurrentMap map = db.hashMap("map").createOrOpen();
map.put("something", "here");
```

HashMap (and other collections) can be also stored in file. In this case the content can be preserved between JVM restarts. It is necessary to call `DB.close()` to protect file from data corruption. Other option is to enable transactions with write ahead log.

```
DB db = DBMaker.fileDB("file.db").make();
ConcurrentMap map = db.hashMap("map").createOrOpen();
map.put("something", "here");
db.close();
```

TODO Hello World examples do not cover the commits.

By default, MapDB uses generic serialization, which can serialize any data type. It is faster and more memory efficient to use specialized serializers. Also we can enable faster memory-mapped files on 64bit operating systems:

```
DB db = DBMaker
        .fileDB("file.db")
        .fileMmapEnable()
        .make();
ConcurrentMap<String,Long> map = db
        .hashMap("map", Serializer.STRING, Serializer.LONG)
        .createOrOpen();
map.put("something", 111L);

db.close();
```

Example projects

TODO example projects

Quick Tips

- Memory mapped files are much faster and should be enabled on 64bit systems for better performance.
- MapDB has Pump for fast bulk import of collections. It is much faster than to `Map.put()`
- Transactions have a performance overhead, but without them the store gets corrupted if not closed properly.
- Data stored in MapDB (keys and values) should be immutable. MapDB serializes objects on background.
- MapDB needs compaction sometimes. Run `DB.compact()` or see background compaction options.

DB and DBMaker

MapDB is pluggable like Lego. There are two classes that act like the glue between the different pieces, namely the `DBMaker` and the `DB` classes.

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

A DB instance represents an opened database (or a single transaction session). It can be used to create and open collection storages. It can also handle the database's lifecycle with methods such as `commit()` , `rollback()` and `close()` .

To open (or create) a store, use one of the many `*DB` static method such as `DBMaker.fileDB()` . MapDB has more formats and modes, whereby each `xxxDB()` uses different modes: `memoryDB()` opens an in-memory database backed by a `byte[]` array, `appendFileDB()` opens a database which uses append-only log files and so on.

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

```
DB db = DBMaker
        .fileDB("/some/file")
        //TODO encryption API
        //.encryptionEnable("password")
        .make();
```

Open and create collection

Once you have DB you may open a collection or other record. DB uses builder style configuration. It starts with type of collection (`hashMap` , `treeSet` ...) and name, followed by configuration is applied and finally by operation indicator

This example opens (or creates new) TreeSet named 'example'

```
NavigableSet treeSet = db.treeSet("example").createOrOpen();
```

You could also apply additional configuration:

```
NavigableSet<String> treeSet = db
        .treeSet("treeSet")
        .maxNodeSize(112)
        .serializer(Serializer.STRING)
        .createOrOpen();
```

The builder can end with three different methods:

- `create()` will create new collection, and throws an exception if collection already exists
- `open()` opens existing collection, and throws an exception if it does not exist
- `createOrOpen()` opens existing collection if it exists, or else creates it.

`DB` is not limited to collections, but creates other type of records such as Atomic Records:

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

Transactions

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

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

```
ConcurrentNavigableMap<Integer,String> map = db
        .treeMap("collectionName", Serializer.INTEGER, Serializer.STRING)
        .createOrOpen();

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();
```

HTreeMap

HTreeMap provides `HashMap` and `HashSet` collections for MapDB. It optionally supports entry expiration and can be used as a cache. It is thread-safe and scales under parallel updates.

It is thread safe, and supports parallel writes by using multiple segments, each with separate ReadWriteLock. `ConcurrentHashMap` in JDK 7 works in a similar way. The number of segments (also called concurrency factor) is configurable.

HTreeMap is a segmented Hash Tree. Unlike other HashMaps it does not use fixed size Hash Table, and does not rehash all data when Hash Table grows. HTreeMap uses auto-expanding Index Tree, so it never needs resize. 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 TODO at what scale?.

HTreeMap optionally supports entry expiration based on four criteria: maximal map size, maximal storage 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.

Serializers

HTreeMap has a number of parameters. Most important is **name**, which identifies Map within DB object and **serializers** which handle data inside Map:

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

//or shorter form
HTreeMap<String, Long> map2 = db
        .hashMap("some_other_map", Serializer.STRING, Serializer.LONG)
        .create();
```

It is also possible to skip serializer definition, but MapDB will use slower generic serialization, and this is not recommended:

```
HTreeMap map = db
        .hashMap("name_of_map")
        .create();
```

HTreeMap is recommended for handling large key/values. In same cases you may want to use compression. It is possible to enable compression store-wide, but that has some 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.hashMap("map")
        .valueSerializer(
                new SerializerCompressionWrapper(Serializer.STRING))
        .create();
```

Hash Code

Most hash maps uses 32bit hash generated by `Object.hashCode()` and check equality with `Object.equals(other)`. However many classes (`byte[]`, `int[]`) do not implement it correctly.

MapDB uses Key Serializer to generate Hash Code and to compare keys. For example `byte[]` can be used directly as key in HTreeMap, if `Serializer.BYTE_ARRAY` is used as Key Serializer:

```
HTreeMap<byte[], Long> map = db.hashMap("map")
        .keySerializer(Serializer.BYTE_ARRAY)
        .valueSerializer(Serializer.LONG)
        .create();
```

Another problem is weak `hashCode()` in some classes, it causes collisions and degrades performance. `String.hashCode()` is weak, but part of specification, so it can not be changed. `HashMap` in JDK implemented many workarounds at the expense of memory and performance overhead. `HTreeMap` has no such workarounds, and weak Hash would slow it down dramatically.

Instead `HTreeMap` is fixing the root of the problem, `Serializer.STRING` uses stronger XXHash which generates less collisions. `String.hashCode()` is still available but with different serializer:

```
//this will use strong XXHash for Strings
HTreeMap<String, Long> map = db.hashMap("map")
        // by default it uses strong XXHash
        .keySerializer(Serializer.STRING)
        .valueSerializer(Serializer.LONG)
        .create();

//this will use weak `String.hashCode()`
HTreeMap<String, Long> map2 = db.hashMap("map2")
        // use weak String.hashCode()
        .keySerializer(Serializer.STRING_ORIGHASH)
        .valueSerializer(Serializer.LONG)
        .create();
```

Hash Maps are vulnerable to Hash Collision Attack. `HTreeMap` adds Hash Seed for protection. It is randomly generated when collection is created and persisted together with its definition. User can also supply its own Hash Seed:

```
HTreeMap<String, Long> map = db
        .hashMap("map", Serializer.STRING, Serializer.LONG)
        .hashSeed(111) //force Hash Seed value
        .create();
```

TODO 64bit Hash Code

TODO custom hash code generator, bit spread, use DataIO.hashInt()

Layout

`HashMap` has parameters such as Initial Capacity, Load Factor etc.. MapDB has different set of parameters to control its access time and maximal size. Those are grouped under term Map Layout.

Concurrency is implemented by using multiple segments, each with separate read-write lock. Each concurrent segment is independent, it has its own Size Counter, iterators and Expiration Queues. Number of segments is configurable. Too small number will cause congestion on concurrent updates, too large will increase memory overhead.

`HTreeMap` uses Index Tree instead of growing `Object[]` for its Hash Table. Index Tree is sparse array like structure, which uses tree hierarchy of arrays. It is sparse, so unused entries do not occupy any space. It does not do rehashing (copy all entries to bigger array), but also it can not grow beyond its initial capacity.

`HTreeMap` layout is controlled by `layout` function. It takes three parameters:

1. **concurrency**, number of segments. Default value is 8, it always rounds-up to power of two.
2. maximal **node size** of Index Tree Dir Node. Default value is 16, it always rounds-up to power of two. Maximal value is 128 entries.
3. number of **Levels** in Index Tree, default value is 4

Maximal Hash Table Size is calculated as: `segment * node size ^ level count`. The default maximal Hash Table Size is `8*16^4=` 0.5 million entries. TODO too low?

If Hash Table Size is set too low, hash collision will start to occur once its filled up and performance will degrade. `HTreeMap` will accept new entries even after Hash Table is full, but performance will degrade.

32bit hash imposes upper limit on Hash Table Size: 4 billion entries. There is a plan to support 64bit hashing.

Other parameters

Another parameter is the **size counter**. By default HTreeMap does not keep track of its size and `map.size()` performs 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
        .hashMap("map", Serializer.STRING, Serializer.LONG)
        .counterEnable()
        .create();
```

And finally some sugar. There is **value loader**, a function to load a value if the existing key is not found. A newly created key/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
        .hashMap("map", Serializer.STRING, Serializer.LONG)
        .valueLoader(s -> 1L)
        .create();

//return 1, even if key does not exist
Long one = map.get("Non Existent");

// Value Creator output was added to Map
map.size(); //  => 1
```

Shard Stores for better concurrency

`HTreeMap` is split into separate segments. Each segment is independent and does not share any state with other segments. However they still share underlying `Store` and that affects performance under concurrent load. It is possible to make segments truly independent, by using separate `Store` for each segment.

That is called **Sharded HTreeMap**, and is created directly `DBMaker`:

```
HTreeMap<String, byte[]> map = DBMaker
         //param is number of Stores (concurrency factor)
        .memoryShardedHashMap(8)
        .keySerializer(Serializer.STRING)
        .valueSerializer(Serializer.BYTE_ARRAY)
        .create();

//DB does not exist, so close map directly
map.close();
```

Sharded HTreeMap has similar configurations options as HTreeMap created by `DB`. But there is no DB object associated with this HTreeMap. So in order to close Sharded HTreeMap, one has to invoke `HTreeMap.close()` method directly.

Expiration

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

- An entry exists in the map longer time than the expiration period is. The expiration period could be since the creation, last modification or since the last read access.
- The number of entries in a map would exceed maximal number
- Map consumes more disk space or memory than space limit

This will set expiration time since the creation, last update and since the last access:

```
// remove entries 10 minutes  after their last modification,
// or 1 minute after last get()
HTreeMap cache = db
        .hashMap("cache")
        .expireAfterUpdate(10, TimeUnit.HOURS)
        .expireAfterCreate(10, TimeUnit.HOURS)
        .expireAfterGet(1, TimeUnit.MINUTES)
        .create();
```

This will create `HTreeMap` with 16GB space limit:

```
// Off-heap map with max size 16GB
Map cache = db
        .hashMap("map")
        .expireStoreSize(16 * 1024*1024*1024)
        .expireAfterGet()
        .create();
```

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

```
HTreeMap cache = db
        .hashMap("cache")
        .expireMaxSize(128)
        .expireAfterGet()
        .create();
```

HTreeMap maintains LIFO Expiration Queue for each segment, eviction traverses queue and removes oldest entries. Not all Map entries are placed into Expiration Queue. For illustration, in this example the new entries never expire, only after update (value change) entry is placed into Expiration Queue.

```
HTreeMap cache = db
        .hashMap("cache")
        .expireAfterUpdate(1000)
        .create();
```

Time based eviction will always place entry into Expiration Queue. But other expiration criteria (size and space limit) also needs hint when to place entry into Expiration Queue. In following example no entry is placed into queue and no entry ever expires.

```
HTreeMap cache = db
        .hashMap("cache")
        .expireMaxSize(1000)
        .create();
```

There are three possible triggers which will place entry into Expiration Queue: `expireAfterCreate()` , `expireAfterUpdate()` and `expireAfterGet()` . Notice there is no TTL parameter.

Entry expiration is done inside other methods. If you call `map.put()` or `map.get()` it might evict some entries. But eviction has some overhead, and it would slow down user operations. There is option to supply HTreeMap with an executor, and perform eviction in background thread. This will evict entries in two background threads, and eviction will be triggered every 10 seconds:

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

ScheduledExecutorService executor =
        Executors.newScheduledThreadPool(2);

HTreeMap cache = db
        .hashMap("cache")
        .expireMaxSize(1000)
        .expireAfterGet()
        .expireExecutor(executor)
        .expireExecutorPeriod(10000)
        .create();

//once we are done, background threads needs to be stopped
db.close();
```

Expiration can be combined with multiple Sharded HTreeMap for better concurrency. In this case each segment has independent Store and that improves scalability under parallel updates.

```
HTreeMap cache = DBMaker
        .memoryShardedHashMap(16)
        .expireAfterUpdate()
        .expireStoreSize(128*1024*1024)
        .create();
```

Sharded HTreeMap should be combined with multiple background threads for eviction. Also over time the Store becomes fragmented and eventually space can not be reclaimed. There is option to schedule periodic compaction if there is too much free space. Compaction will reclaim free space. Because each Store (segment) is compacted separately, compactions do not affect all running threads.

```
HTreeMap cache = DBMaker
        .memoryShardedHashMap(16)
        .expireAfterUpdate()
        .expireStoreSize(128*1024*1024)

        //entry expiration in 3 background threads
        .expireExecutor(
                Executors.newScheduledThreadPool(3))

        //trigger Store compaction if 40% of space is free
        .expireCompactThreshold(0.4)

        .create();
```

Expiration Overflow

HTreeMap supports Modification Listeners. It notifies listener about inserts, updates and removes from HTreeMap. It is possible to link two collections together. Usually faster in-memory with limited size, and slower on-disk with unlimited size. After an entry expires from in-memory, it is automatically moved to on-disk by Modification Listener. And Value Loader will load values back to in-memory map, if those are not found by map.get() operation.

To establish Disk Overflow use following code:

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

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

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

// fast in-memory collection with limited size
HTreeMap inMemory = dbMemory
        .hashMap("inMemory")
        .expireAfterGet(1, TimeUnit.SECONDS)
        //this registers overflow to `onDisk`
        .expireOverflow(onDisk)
        //good idea is to enable background expiration
        .expireExecutor(Executors.newScheduledThreadPool(2))
        .create();
```

Once binding is established, every entry removed from `inMemory` map will be added to `onDisk` map. This applies only to expired entries, `map.remove()` will also remove any entry from `onDisk` .

```
//insert entry manually into both maps for demonstration
inMemory.put("key", "map");

//first remove from inMemory
inMemory.remove("key");
onDisk.get("key"); // -> not found
```

If the `inMemory.get(key)` is called and value does not exist, the Value Loader will try to find Map in `onDisk` . If value is found inside `onDisk` , it is added 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
```

It is also possible to clear entire primary map and move all data to disk:

```
inMemory.put(1,11);
inMemory.put(2,11);

//expire entire content of inMemory Map
inMemory.clearWithExpire();
```

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

TODO modification listeners

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

Parameters

BTreeMap has optional parameters which can be specified with the use of a 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.treeMap("map")
        .keySerializer(Serializer.LONG)
        .valueSerializer(Serializer.STRING)
        .createOrOpen();
```

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

BTrees store all their keys and values as part of a btree node. Node size affects the 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
        .treeMap("map", Serializer.LONG, Serializer.STRING)
        .counterEnable()
        .createOrOpen();
```

Values are also stored as part of BTree leaf nodes. Large value means huge overhead and on a 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, the leaf node only has a 6 byte recid pointing to the value.

Large values can also be compressed to save space. This example stores values outside BTree Leaf Node and applies compression on each value:

```
BTreeMap<Long, String> map = db.treeMap("map")
        .valuesOutsideNodesEnable()
        .valueSerializer(new SerializerCompressionWrapper(Serializer.STRING))
        .createOrOpen();
```

BTreeMap needs to sort its key somehow. By default it relies on the `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.treeMap("map")
        // use array serializer for unknown objects
        // TODO db.getDefaultSerializer()
        .keySerializer(new SerializerArray(Serializer.JAVA))
        // or use serializer for specific objects such as String
        .keySerializer(new SerializerArray(Serializer.STRING))
        .createOrOpen();
```

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.treeMap("map")
        .keySerializer(Serializer.BYTE_ARRAY)
        .valueSerializer(Serializer.LONG)
        .createOrOpen();
```

Key serializers

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

A long key occupies 8 bytes after serialization. To minimize the 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` the keys are stored in awrapped form, such as `Long[]`. That has a huge overhead, as each key requires a 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 static fields with `_PACK` suffix in Serializer class for more details.

TODO this is a major feature, document details and add benchmarks

Data Pump

TODO data pump

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 the content into a new `BTreeMap`. As it is very fast with Data Pump streaming, the 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 second small map will contain updates. Periodically a third map will be produced as a merge of the first two, and will be swapped with the primary. `SSTable`'s in Cassandra and other databases work in a similar way.

TODO provide wrapper to compact/merge BTreeMap content automatically.

Prefix submaps

For array based keys (tuples, Strings, or arrays) MapDB provides prefix submap. It uses intervals, so prefix submap is lazy, it does not load all keys. Here as example which uses prefix on `byte[]` keys:

```
BTreeMap<byte[], Integer> map = db
        .treeMap("towns", Serializer.BYTE_ARRAY, Serializer.INTEGER)
        .createOrOpen();

map.put("New York".getBytes(), 1);
map.put("New Jersey".getBytes(), 2);
map.put("Boston".getBytes(), 3);

//get all New* cities
Map<byte[], Integer> newCities = map.prefixSubMap("New".getBytes());
```

TODO key serializer must provide `nextValue` for prefix submaps. Implement it on more serializers

Composite keys and tuples

MapDB allows composite keys in the form of `Object[]`. Interval submaps can be used to fetch tuple subcomponents, or to create a simple form of multimap. Object array is not comparable, so you need to use specialized serializer which provides comparator.

Here is an example which creates `Map<Tuple3<String, String, Integer>, Double>` in the form of Object[]. First component is town, second is street and third component is house number. It has more parts, source code is on github To serialize and compare tuples use `SerializerArrayTuple` which takes serializer for each tuple component as s constructor parameter:

```
// initialize db and map
DB db = DBMaker.memoryDB().make();
BTreeMap<Object[], Integer> map = db.treeMap("towns")
        .keySerializer(new SerializerArrayTuple(
                Serializer.STRING, Serializer.STRING, Serializer.INTEGER))
        .valueSerializer(Serializer.INTEGER)
        .createOrOpen();
```

Once map is populated we can get all houses in the town of Cong by using prefix submap (town is first component in tuple):

```
//get all houses in Cong (town is primary component in tuple)
Map<Object[], Integer> cong =
        map.prefixSubMap(new Object[]{"Cong"});
```

The prefix submap is equal to the range query which uses submap method:

Interval submap can only filter components on the left side. To get components in the middle we have to combine the submap with a forloop:

```
cong = map.subMap(
        new Object[]{"Cong"},           //shorter array is 'negative infinity'
        new Object[]{"Cong",null,null} // null is positive infinity'
);
```

Submaps are modifiable, so we could delete all the houses within a town by calling clear() on submap etc..

Multimap

Multimap is a Map which associates multiple values with a single key. An example can be found in Guava or in Eclipse Collections It can be written as Map<Key,List<Value>>, but that does not work well in MapDB, we need keys and values to be immutable, and List is not immutable.

There is a plan to implement Multimap from Guava and EC directly in MapDB. But until then there is an option to use SortedSet in combination with tuples and interval subsets. Here is an example which constructs Set, inserts some data and gets all values (second tuple component) associated with key (first tuple component):

```
// initialize multimap: Map<String,List<Integer>>
NavigableSet<Object[]> multimap = db.treeSet("towns")
        //set tuple serializer
        .serializer(new SerializerArrayTuple(Serializer.STRING, Serializer.INTEGER))
        .counterEnable()
        .counterEnable()
        .counterEnable()
        .createOrOpen();

// populate, key is first component in tuple (array), value is second
multimap.add(new Object[]{"John",1});
multimap.add(new Object[]{"John",2});
multimap.add(new Object[]{"Anna",1});

// print all values associated with John:
Set johnSubset = multimap.subSet(
        new Object[]{"John"},         // lower interval bound
        new Object[]{"John", null});  // upper interval bound, null is positive infini
ty
```

TODO delta packing for Tuples

TODO MapDB will soon implement multimap from Guava

Compared to HTreeMap

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

TODO compare to HTreeMap

Composite Keys

BTreeMap can have composite key; an key composed from multiple components. Range query can get all sub-components associated with primary component.

Here is an example; lets associate persons name (composed of surname and firstname) with age. We than find all persons with surname Smith (primary key component) .

```
    //create new map
    BTreeMap<Tuple2, Integer> persons = db
      .treeMap("persons", Tuple2.class, Integer.class)
      .createOrOpen();

    //insert three persons into map
    persons.put(new Tuple2("Smith","John"), 45);
    persons.put(new Tuple2("Smith","Peter"), 37);
    persons.put(new Tuple2("Doe","John"), 70);

    //now lets get map which contains all Smiths
    NavigableMap<Tuple2,Integer> smiths =
      persons.prefixSubMap(
new Tuple2("Smith", null)  //null indicates wildcard for range query
      );
```

Example above can be more strongly-typed with wrapper classes and generics. In here we use `Surname` and `Firstname` classes.

```
    //create new map
    BTreeMap<Tuple2<Surname, Firstname>, Integer> persons = db
      .treeMap("persons")
      .keySerializer(new Tuple2Serializer()) //specialized tuple serializer
      .valueSerializer(Serializer.INTEGER)
      .createOrOpen();

    //insert three person into map
    persons.put(new Tuple2(new Surname("Smith"),new Firstname("John")), 45);
    persons.put(new Tuple2(new Surname("Smith"),new Firstname("Peter")), 37);
    persons.put(new Tuple2(new Surname("Doe"),new Firstname("John")), 70);

    //now lets get map which contains all Smiths
    NavigableMap<Tuple2<Surname, Firstname>,Integer> smiths =
      persons.prefixSubMap(
new Tuple2(new Surname("Smith"), null)  //null indicates
      );
```

Tuples use `Comparable` interface, all key components (`Person` and `Firstname`) should implement it. Other option is to use composite serializer with comparator method. For example to have `Tuple2<byte[], byte[]>` key we create tuple serializer following way: `new Tuple2Serializer(Serializer.BYTE_ARRAY, Serializer.BYTE_ARRAY)` . Complete example:

```
    //create new map
    BTreeMap<Tuple2<byte[], byte[]>, Integer> persons = db
      .treeMap("persons")
      .keySerializer(new Tuple2Serializer(Serializer.BYTE_ARRAY, Serializer.BYTE_ARRAY
))
      .valueSerializer(Serializer.INTEGER)
      .createOrOpen();

    persons.put(new Tuple2("Smith".getBytes(),"John".getBytes()), 45);

    NavigableMap<Tuple2,Integer> smiths =
      persons.prefixSubMap(
new Tuple2("Smith".getBytes(), null)
      );
```

Range query

In examples above we used `prefixSubMap(new Tuple2("surname", null))` method. It performs range query where second component is replaced by minimal and maximal value. This method `BTreeMap` class and is not standard `Map` method, there is `NavigableMap.subMap` equivalent:

```
    NavigableMap<Tuple2,Integer> smiths =
      persons.prefixSubMap(new Tuple2("Smith", null));

    // is equivalent to
    smiths = persons.subMap(
      new Tuple2("Smith", Integer.MIN_VALUE), true,
      new Tuple2("Smith", Integer.MAX_VALUE), true
    );
```

In example above we use `Integer` because it provides minimal and maximal values. To make this easier `TupleSerializer` introduces special values for negative and positive infinity, those are even smaller/greater than min/max values. `null` corresponds to negative infinity, `Tuple.HI` is positibe infinity.

Those two values are not serializable and can not be stored in Map. But can be used for range query:

```
persons.subMap(
  new Tuple2("Smith", null),
  new Tuple2("Smith", Tuple.HI)
);
```

Submap returns only single range. It means that we can only query left most components. Common mistake is to put infinity in middle, and expect right components to be included. Tuple in example bellow has three components (surname, firstname, age). But we can not just query Surname and Age, because age is left most and it will be overriden by infinity component before it:

```
//WRONG!! null is in middle position
persons.prefixSubMap(new Tuple3("Smith",null,11));

//same but submap
//WRONG!! infinity is in middle
persons.subMap(
  new Tuple3("Smith", null,     11),
  new Tuple3("Smith", Tuple.HI, 11)
);
```

Fixed size array tuples

Tuples can be replaced by array. In this case we do not have generics and will have to do lot of casting. Here is an example with surname/firstname. For key serializer we use `new SerializerArrayTuple(tupleSize)`. `null` and `Tuple.HI` will not work, but we can use shorter array for prefix:

```
//create new map
BTreeMap<Object[], Integer> persons = db
  .treeMap("persons", new SerializerArrayTuple(2), Serializer.INTEGER)
  .createOrOpen();

//insert three person into map
persons.put(new Object[]{"Smith", "John"}, 45);
persons.put(new Object[]{"Smith", "Peter"}, 37);
persons.put(new Object[]{"Doe", "John"}, 70);

//now lets get map which contains all Smiths
NavigableMap<Object[],Integer> smiths =
  persons.prefixSubMap(
new Object[]{"Smith"}
  );
```

//TODO null is positive infinity, Tuple.HI does not exist

Variable size array tuples

MapDB also has generic array serializer which can be used for tuples. In this case `prefixSubmap` will not work. But we can use submap:

```
    //create new map
    BTreeMap<Object[], Integer> persons = db
      .treeMap("persons", new SerializerArrayDelta(), Serializer.INTEGER)
      .createOrOpen();

    //insert three persons into map
    persons.put(new Object[]{"Smith", "John"}, 45);
    persons.put(new Object[]{"Smith", "Peter"}, 37);
    persons.put(new Object[]{"Doe", "John"}, 70);


    NavigableMap<Object[],Integer> smiths = persons.subMap(
      new Object[]{"Smith"}, //lower bound
      new Object[]{"Smith", null} //upper bound, null is positive infinity in this ser
ializer
    );
```

Delta compression

All three tuples type use delta compression.

TODO delta compression

Batch Import in BTreeMap

BTreeMap (as any other b-tree) suffers from write amplification. Single entry update has to traverse tree, and modify entire tree node. Inserting many entries might be too slow.

There are two solutions for this problem. Write cache improves write amplification if updated entries are near each other. Tree node is updated many times in cache, but written only once when cache is flushed.

Another solution is to import BTreeMap from sorted stream (in older versions called Data Pump). It takes sorted stream of entries, and creates tree structure directly from stream.

Stream import does not use random IO, only sequential write. Nodes are never modified, only created. So in practice it imports BTreeMap at rate 50 MB/s. Also import speed does not degrade as btree becomes larger (`N*log(N)`), it can create multi-TB b-trees on spinning disk in just a few hours.

Only downside is that imported data needs to be **sorted in ascending order**. Older MapDB version required data sorted in reversed descending order, that is solved in 3.0.

Here we create TreeSet from sorted iterator:

```
        // note that source data are sorted
        List<Integer> source = Arrays.asList(1,2,3,4,5,7,8);

        //create map with content from source
        NavigableSet<Integer> set = db.treeSet("set")
  .serializer(Serializer.INTEGER)
  .createFrom(source); //use `createFrom` instead of `create`
```

It is also possible to import SortedMap from an iterator. In this case we need to use iterator of `Pair(key,value)` , or another sorted Map as a source.

```
      // source data, first entry in pair is key, second is value
      // note that source data are sorted
      List<Pair<Integer,Integer>> source =
  Arrays.asList(new Pair(1,2),new Pair(3,4),new Pair(5,7));

      //create map with content from source
      BTreeMap<Integer, Integer> map = db.treeMap("map")
.keySerializer(Serializer.INTEGER)
.valueSerializer(Serializer.INTEGER)
.createFrom(source); //use `createFrom` instead of `create`

      //we can also use another source map as a source
      db.treeMap("map2")
.keySerializer(Serializer.INTEGER)
.valueSerializer(Serializer.INTEGER)
.createFrom(map);
```

Create using Sink

Some data are not available in collections or as an iterator, for example when you are receive data in packets or read file line by line. For that case MapDB provides a Sink, an callback class which accepts entries and has finish method. Here is an example which fills Map using for-loop:

```
      //create sink
      DB.TreeMapSink<Integer,String> sink = db
.treeMap("map", Serializer.INTEGER,Serializer.STRING)
.createFromSink();

      //loop and pass data into map
      for(int lineNum=0;lineNum<10000;lineNum++){
String line = "some text from file"+lineNum;
//add key and value into sink, keys must be added in ascending order
sink.put(lineNum, line);
      }
      // Sink is populated, map was created on background
      // Close sink and return populated map
      BTreeMap<Integer,String> map = sink.create();
```

Sorted Table Map

`SortedTableMap` is inspired by Sorted String Tables from Cassandra. It stores keys in file (or memory store) in fixed size table, and uses binary search. There are some tricks to support variable-length entries and to decrease space usage. Compared to `BTreeMap` it is faster, has zero fragmentation, but is readonly.

`SortedTableMap` is read-only and does not support updates. Changes should be applied by creating new Map with Data Pump. Usually one places change into secondary map, and periodically merges two maps into new `SortedTableMap` .

`SortedTableMap` is read-only. Its created and filled with content by Data Pump and Consumer:

```
//create memory mapped volume
Volume volume = MappedFileVol.FACTORY.makeVolume(file, false);

//open consumer which will feed map with content
SortedTableMap.Sink<Integer,String> sink =
        SortedTableMap.create(
                volume,
                Serializer.INTEGER,
                Serializer.STRING
        ).createFromSink();

//feed content into consumer
for(int key=0; key<100000; key++){
    sink.put(key, "value"+key);
}

// finally open created map
SortedTableMap<Integer, String> map = sink.create();
```

Once file is created, it can be reopened:

```
//open existing  memory-mapped file in read-only mode
Volume volume = MappedFileVol.FACTORY.makeVolume(file, true);
                                                         //read-only=true

SortedTableMap<Integer,String> map =
        SortedTableMap.open(
                volume,
                Serializer.INTEGER,
                Serializer.STRING
                );
```

Binary search

Storage is split into pages. Page size is power of two, with maximal size 1MB. First key on each page is stored on-heap.

Each page contains several nodes composed of keys and values. Those are very similar to BTreeMap Leaf nodes. Node offsets are known, so fast seek to beginning of node is used.

Each node contains several key-value pairs (by default 32). Their organization depends on serializer, but typically are compressed together (delta compression, LZF..) to save space. So to find single entry, one has to load/traverse entire node. Some fixed-length serializer (Serializer.LONG...) do not have to load entire node to find single entry.

Binary search on `SortedTableMap` is performed in three steps:

- First key for each page is stored on-heap in an array. So perform binary search to find page.
- First key on each node can by loaded without decompressing entire node. So perform binary search over first keys on each node
- Now we know node, so perform binary search over node keys. This depends on Serializer. Usually entire node is loaded, but other options are possible TODO link to serializer binary search.

Parameters

`SortedTableMap` takes key **serializer** and value serializers. The keys and values are stored together inside Value Array TODO link to serializers. They can be compressed together to save space. Serializer is trade-off between space usage and performance.

Another setting is **Page Size**. Default and maximal value is 1MB. Its value must be power of two, other values are rounded up to nearest power of two. Smaller value typically means faster access. But for each page one key is stored on-heap, smaller Page Size also means larger memory usage.

And finally there is **Node Size**. It has similar implications as BTreeMap node size. Larger node means better compression, since large chunks are better compressible. But it also means slower access times, since more entries are loaded to get single entry. Default node size is 32 entries, it should be lowered for large values.

Parameters are set following way

```
//create memory mapped volume
Volume volume = MappedFileVol.FACTORY.makeVolume(file, false);

//open consumer which will feed map with content
SortedTableMap.Sink<Integer,String> sink =
        SortedTableMap.create(
                volume,
                Serializer.INTEGER, // key serializer
                Serializer.STRING   // value serializer
        )
                .pageSize(64*1024) // set Page Size to 64KB
                .nodeSize(8)       // set Node Size to 8 entries
                .createFromSink();

//feed content into consumer
for(int key=0; key<100000; key++){
    sink.put(key, "value"+key);
}

// finally open created map
SortedTableMap<Integer, String> map = sink.create();
volume.close();

// Existing SortedTableMap can be reopened.
// In that case only Serializers needs to be set,
// other params are stored in file
volume = MappedFileVol.FACTORY.makeVolume(file, true);
                                                     // read-only=true
map = SortedTableMap.open(volume, Serializer.INTEGER, Serializer.STRING);
```

Volume

`SortedTableMap` does not use `DB` object, but operates directly on `Volume` (MapDB abstraction over ByteBuffer). Following example show how to construct various `Volume` using in-memory byte array or memory-mapped file:

```
//create in-memory volume over byte[]
Volume byteArrayVolume = ByteArrayVol.FACTORY.makeVolume(null, false);

//create in-memory volume in direct memory using DirectByteByffer
Volume offHeapVolume = ByteBufferMemoryVol.FACTORY.makeVolume(null, false);

File file = File.createTempFile("mapdb","mapdb");
//create memory mapped file volume
Volume mmapVolume = MappedFileVol.FACTORY.makeVolume(file.getPath(), false);

//or if data were already imported, create it read-only
mmapVolume.close();
mmapVolume = MappedFileVol.FACTORY.makeVolume(file.getPath(), true);
                                                                    //read-only=true
```

Volume is than passed to `SortecTableMap` factory method as an parameter. It is recommended to open existing Volumes in read-only mode (last param is `true`) to minimize file locking and simplify your code.

Data Pump sync Volume content to disk, so file based `SortedTableMap` is durable once the `Consumer.finish()` method exits

Performance and durability

Good performance is result of compromise between consistency, speed and durability. MapDB gives several options to make this compromise. There are different storage implementations, commit and disk sync strategies, caches, compressions...

This chapter outlines performance and durability related options. Some options will make storage writes durable at expense of speed. Some other settings might cause memory leaks, data corruption or even JVM crash! Make sure you understand implications and read Javadoc on DBMaker.

Transactions and crash protection

If store is not closed properly and are pending changes flushed to disk, store might become corrupted. That often happens if JVM process crashes or is violently terminated.

To protect file from corruption, MapDB offers Write Ahead Log (WAL). It is reliable and simple way to make file changes atomic and durable. WAL is used by many databases including Posgresql or MySQL. However WAL is slower, data has to be copied and synced multiple times between files.

WAL is disabled by default. It can be enabled with `DBMaker.transactionEnable()` :

```
DB db = DBMaker
        .fileDB(file)
        .transactionEnable()
        .make();
```

With WAL disabled (by default) you do not have a crash protection. In this case you **must** correctly close the store, or you will loose all your data. MapDB detects unclean shutdown and will refuse to open such corrupted storage. There is a way to open corrupted store in readonly mode and perform data rescue.

There is a shutdown hook to close the database automatically before JVM exits, however this does not protect your data if JVM crashes or is killed. Use `DBMaker.closeOnJvmShutdown()` option to enable it.

With transactions disabled you do not have 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.

TODO JVM write cache flush, versus system flush.

Memory mapped files (mmap)

MapDB was designed from ground to take advantage of mmap files. However on 32bit JVM mmap files are limited to 4GB by its addressing limit. When JVM runs out of addressing space there are nasty effects such as JVM crash. 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.

There is also bug in JVM. Mmaped file handles are not released until `DirectByteBuffer` is GCed. That means that mmap file remains open even after `db.close()` is called. On Windows it prevents file to be reopened or deleted. On Linux it consumes file descriptors, and could lead to errors once all descriptors are used.

There is a workaround for this bug using undocumented API. But it was linked to JVM crashes in rare cases and is disabled by default. Use `DBMaker.cleanerHackEnable()` to enable it.

Here is example with all mmap related options:

```
DB db = DBMaker
    .fileDB(file)
    .fileMmapEnable()            // Always enable mmap
    .fileMmapEnableIfSupported() // Only enable mmap on supported platforms
    .fileMmapPreclearDisable()   // Make mmap file faster

        // Unmap (release resources) file when its closed.
        // That can cause JVM crash if file is accessed after it was unmapped
        // (there is possible race condition).
    .cleanerHackEnable()
    .make();

//optionally preload file content into disk cache
db.getStore().fileLoad();
```

File channel

By default MapDB uses `RandomAccessFile` to access disk storage. Outside fast mmap files there is third option based on `FileChannel`. It should be faster than `RandomAccessFile`, but has bit more overhead. It also works better under concurrent access (RAF has global lock).

FileChannel was causing problems in combination with `Thread.interrupt`. If threads gets interrupted while doing IO, underlying channel is closed for all other threads.

To use FileChannel use `DBMaker.fileChannelEnable()` option:

```
DB db = DBMaker
    .fileDB(file)
    .fileChannelEnable()
    .make();
```

In-memory mode

MapDB has three in-memory stores:

On-heap which stores objects in `Map<recid,Object>` and does not use serialization. This mode is very fast for small datasets, but is affected by GC, so performance drops from cliff after a few gigabytes. It is activated with:

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

Store based on `byte[]`. In this mode data are serialized and stored into 1MB large byte[]. Technically this is still on-heap, but is not affected by GC overhead, since data are not visible to GC. This mode is recommended by default, since it does not require any additional JVM settings. Increasing maximal heap memory with `-Xmx10G` JVM parameter is enough.

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

Store based on `DirectByteBuffer`. In this case data are stored completely off-heap. in 1MB DirectByteBuffers created with `ByteBuffer.allocateDirect(size)`. You should increase maximal direct memory with JVM parameter. This mode allows you to decrease maximal heap size to very small size (`-Xmx128M`). Small heap size has usually better and more predictable performance.

```
// run with: java -XX:MaxDirectMemorySize=10G
DB db = DBMaker
    .memoryDirectDB()
    .make();
```

Allocation options

By default MapDB tries minimize space usage and allocates space in 1MB increments. This additional allocations might be slower than single large allocation. There are two options to control storage initial size and size increment. This example will allocate 10GB initially and then increment size in 512MB chunks:

```
DB db = DBMaker
    .fileDB(file)
    .fileMmapEnable()
    .allocateStartSize( 10 * 1024*1024*1024)  // 10GB
    .allocateIncrement(512 * 1024*1024)       // 512MB
    .make();
```

Allocation Increment has side effect on performance with mmap files. MapDB maps file in series of DirectByteBuffer. Size of each buffer is equal to Size Increment (1MB by default), so larger Size Increment means less buffers for the same disk store size. Operations such as sync, flush and close have to traverse all buffers. So larger Size Increment could speedup commit and close operations.

Collection Layout

Partitioning in databases is usually way to distribute data between multiple stores, tables etc... MapDB has great flexibility and its partitioning is more complicated. So in MapDB partitioning is when collection is using more than single Store, to contain its state.

For example `HTreeMap` can split key-value entries between multiple disks, while its Hash Table uses in-memory Store and Expiration Queues are regular on-heap collections.

This chapter gives overview of most partitioning options in MapDB. Details are in separate chapters for each collection or Store.

Hash Partitioning

HP is well supported in HTreeMap *TODO link*. To achieve concurrency HashMap is split into segments, each segment is separate HashMap with its own ReadWriteLock. Segment number is calculated from hash. When expiration is enabled each segment has its own Expiration Queue.

Usually HTreeMap segments share single Store *TODO link*. But each segment can have its own Store, that improves concurrency and allows to shard HTreeMap across multiple disks.

Range partitioning

Is not currently supported in BTreeMap.

TODO discus sharding based on Node Recid hash

TODO investigate feasibility in BTreeMap

Time of Update Partitioning

Large BTrees usually has slow updates, due to write amplification *TODO chapter & link*. In some cases (time series) it makes sense to shard data based on last modification. Each day (or other interval) has its own store, old data can be removed just by deleting files. There are various options to handle modifications, delete markers etc... MapDB supports this with SortedTableMap and CopyOnWriteMap

TODO this could also work on HTreeMap (sorted by hash)

Partitioning by Durability

Durable commits are much slower than non-durable. We have to move data to/from write-ahead-log, sync files, calculate checksums... Durability Partitioning allows to minimize size of durable data, by moving non essential data into non-durable store. Trade off is longer recovery time after crash.

Good example is BTree. We really only care about Leaf Nodes, which contains all key-value pairs. Directory nodes (index) can be easily reconstructed from Leaf Nodes. BTreeMap can use two stores, one with durable commits for leafs, second non-durable for directory nodes. Pump than reconstructs Directory Store in case of crash.

Name?

TODO HTreeMap expiration queues onheap, in memory

TODO in-memory indexes for HTreeMap and BTreeMap

TODO HTreeMap IndexTree onheap

Expiration Partitioning

TODO HTreeMap disk overflow

Storage format

This chapter is storage specification for MapDB files.

File operations

File operations (such as file create, rename or sync) must be atomic and must survive system crash. In case of crash there is recovery operation after restart. If file operation did not finished it reverts everything into last stable state. That means file operations are atomic (they either succeed or fail without side effects).

To ensure crash resistance and atomicity MapDB relies on marker files. Those are empty files created and deleted using atomic filesystem API. Marker files have the same name as main file, but with `.$X` suffix.

File Create

Empty file creation is atomic operation, but populating file with content is not. MapDB needs file population to be atomic, and uses uses `.$c` marker file for that.

File creation sequence:

1. create marker file with `File.createNewFile()`
2. create empty main file and lock it
3. fill main file with content, write checksums
4. sync main file
5. remove marker file

In case of recovery, or when file is being opened, follow this sequence:

1. open main file and lock it, fail if main file does not exist
2. TODO we should check for pending File Rename operations here
3. check if marker file exists, fail if it exists

In case of failure throw an data corruption exception.

Temporary file write open

Temporary file in MapDB is write-able file without crash protection (usually by write-ahead-log). Compared to *File Create* this file is opened continuously and only closed on system shutdown. If file was not closed, it most likely becomes corrupted and MapDB will refuse to reopen in.

File Create sequence is also used for temporary file without crash protection. In that case marker file stays while the main file is opened for write. If there is an crash, recovery sequence will find marker file, assume that main file was not closed correctly and will refuse to open it. In this case main file should be discarded and recreated from original data source. Or user can remove marker file and try his luck.

File Rename

File Rename is used in StoreDirect compaction. Store is recreated in new file, and old file is replaced with new content. The 'old file' is file which is being replaced, it will be deleted before File Rename. The 'new file' replaces old file and has its name changed.

MapDB needs file move to be atomic, and supported in range variety of platforms. There are following problems:

- `java.nio.file.Files#move` is atomic, but it might fail in some cases
- Opened memory mapped file on Windows can not be renamed. MappedByteBuffer handle is not released until GC or cleaner hack. Sometimes handle is not released even after JVM exit, and OS restart is required.
- There should be fallback option, when we can not close file Volume, but copy content between Volumes.

File rename has following sequence:

- synchronize and close new file, release its c marker
- create 'c' marker on old file
- create 'r' marker on new file
- delete old file
- use `java.nio.file.Files#move` in atomic or non-atomic way. But rename operation must be finished and synced to disk.
- delete r marker for new file
- delete c marker on old file
- open old file (with new content)

TODO this does not work on windows with memory mapped files. We need plan B with Volume copy, without closing them.

Recovery sequence is simple. If following files exist:

- c marker for old file
- r marker for new file
- new file (under its name before rename)

Than discard the old file if present and continue rename sequence from step 'delete old file'

Rolling file

Rolling file is a single file, but continuously replaced with new content. To make content replacement atomic, the content of file is written into new file, synced and then old file is deleted. File name has '.N' suffix, where N is sequential number increased with each commit. Rolling file is used in `StoreTrivial`.

There is following sequence for updating rolling file with new content. Ther is 'old file' with original content and number N and 'new file' with number N+1.

- Create c marker for new file, fail if it already exists
- Populate new file with content, sync and close
- Remove C marker for new file
- Delete the old file

And there is following sequence for recovery

- List all files in parent directory, find file with highest number without C marker, lock and open it.
- Delete any other files and their markers (only files associated with the rolling file, there might be more files with different name)

File sync

On commit or close, write cache needs to be flushed to disk, in MapDB this is called sync. We also need to detect corrupted files if system crashes in middle of write.

There are following ways to sync file:

- **'c' file marker** (see File Rename).
- **File checksum:** Before the file sync is called, checksum of entire file is calculated and written into file header. Corruption is detected by matching file checksum from header with file content. This is slow because entire file has to be read
- **Commit seal:** Uses double file sync, but does not require checksum calculation. First file is synced with zero checksum in file header. Than commit seal is written into file header, and file is synced again. Valid commit seal means that file was synced. TODO: commit seal is calculated based on file size

File header

Every non empty file created by MapDB has 16 byte header. It contains header, file version, bitfield for optional features and optional checksum for entire file.

Bites:

- 0-7 constant value 0x4A
- 8-15 type of file generated. I
- 16-31 format version number. File will not be opened if format is too high
- 32-63 bitfield which identifies optional features used in this format. File will not be opened if unknown bit is set.
- 64-127 checksum of entire file.

File type

can have following values:

- 0 unused
- 1 StoreDirect (also shared with StoreWAL)
- 2 WriteAheadLog for StoreWAL
- 10 SortedTableMap without multiple tables (readonly)
- 11 SortedTableMap with multiple tables
- 12 WriteAheadLog for SortedTableMap

Feature bitfield

has following values. It is 8-byte long, number here is from least significant bit.

- 0 encryption enabled. Its upto user to provide encryption type and password
- 1-2 checksum used. 0=no checksum, 1=XXHash, 2=CRC32, 3=user hash.
- TODO more bitfields

Checksum

is either XXHash or CRC32. It is calculated as `(checksum from 16th byte to end)+vol.getLong(0)`. If checksum is `0` the `1` value is used instead. `0` indicates checksum is disabled.

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 parity bits as passive protection from returning 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.

Head

Header in StoreDirect format is composed by number of 8-byte longs. Each offset here is multiplied by 8

1. header and format version from file header *TODO chapter link*
2. file checksum from file header *TODO chapter link*
3. **header checksum** is updated every time header is modified, that can detect corruption quite fast
4. **data tail** points to end location where data were written to. Beyond this is empty (except index pages). Parity 4 with no shift (data offset is multiple of 16)
5. **max recid** maximal allocated recid. Parity 4 with shift.
6. **file tail** file size. Must be multiple of PAGE_SIZE (1MB). Parity 16
7. not yet used
8. not yet used

This is followed by Long Stack Master Pointers. Those are used to track free space, unused recids and other information.

- `8` - **Free recid** Long Stack, unused Recids are put here
- `9` - **Free records 16** - Long Stack with offsets of free records with size 16
- `10` - **Free records 32** - Long Stack with offsets of free records with size 32 etc...
- ...snip 4095 minus 3 entries...
- `8+4095` - **Free records 65520** - Long Stack with offsets of free records with size 65520 bytes (maximal unlinked record size). 4095 = 65520/16 is number of Free records Long Stacks.
- `8+4095+1` until `8+4095+4` - **Unused long stacks** - Those could be used latter for some other purpose.

Index page

Rest of the zero page (up to offset 1024*1024) is used as Index Page (sometimes it is refered as Zero Index Page). Offset to next Index Page (First Index Page) is at `8+4095+4+1`, Zero Index Page checksum is at `8+4095+4+2`. First recid value is at `8+4095+4+3`.

Index page starts at `N*PAGE_SIZE`, except Zero Index Page which starts at `8 * (8+4095 + 4 + 1)`. Index page contains at start:

- zero value (offset `page+0`) is **pointer to next index page**, Parity 16
- first value (offset `page+8`) in page is **checksum of all values** on page (add all values) *TODO seed? and not implemented yet*

Rest of the index page is filled with index values.

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

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 aligned 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.

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

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.

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

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

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:

1. **end of file**. Last instruction of file. Checksum is `bit parity from offset & 31`
2. **write long**. Is followed by 8 bytes value and 6 byte offset. Checksum is `(bit count from 15 bytes + 1)&31`
3. **write byte[]**. Is followed by 2 bytes size, 6 byte offset and data itself. Checksum is `(bit count from size + bit count from offset + 1 )&31`
4. **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 count from 3 bytes + 1)&31`
5. **skip single byte**. Skip single byte in WAL. Checksum is `bit count from offset & 31`
6. **record**. Is followed by packed recid, than packed record size and an record data. Real size is +1, 0 indicates null record TODO checksum for record inst
7. **tombstone**. Is followed ba packed recid. . Checksum is `bit count from offset & 31`
8. **preallocate**. Is followed ba packed recid. . Checksum is `bit count from offset & 31`
9. **commit**. TODO checksum
10. **rollback**. TODO checksum

Sorted Table Map

`SortedTableMap` uses its own file format. File is split into page, where page size is power of two and maximal page size 1MB.

Each page has header. Header size is bigger for zero page, because it also contains file header. TODO header size.

After header there is a series of 4-byte integers.

First integer is number of nodes on this page (N). It is followed by N*2 integers. First N integers are offsets of key arrays for each node. Next N integers are offsets for value arrays for each node. Offsets are relative to page offset. The last integer points to end of data, rest of the page after that offset is filled with zeroes.

Offsets of key array (number i) are stored at: `PAGE_OFFSET + HEAD_SIZE + I*4`.

Offsets of value array (number i) are stored at: `PAGE_OFFSET + HEAD_SIZE + (NODE_COUNT + I) * 4`.