MapDB Cheat Sheet

www.mapdb.org

Maven

```
<dependency>
   <groupId>org.mapdb</groupId>
   <artifactId>mapdb</artifactId>
   <version>[version]</version>
</dependency>
```

Map stored in file

```
import org.mapdb.*;

//open (or create) database
File file = new File(“dbFileName”);
DB db = DBMaker
        .newFileDB(file)
        .make();

//use map
Map map = db.getHashMap(“mapName”);
map.put(“aa”,”bb”);

//commit and close database
db.commit();
db.close();
```

In-memory off-heap Map

```
// same as above except different method
DB db = DBMaker
        .newMemoryDirectDB();
        .make();
```

In-memory off-heap Queue

```
// same as above except different method
DB db = DBMaker
        .newMemoryDirectDB();
        .make();
Queue q = db.getQueue(“q”);
```

Options to make it faster

```
DB db = DBMaker
// all options works with files as well
  .newMemoryDB();
// disable transactions make writes
// but you may lose data if store crashes
  .transactionsDisable()
// memory mapped files are faster
// but does not work well on 32bit
  .mmapFileEnable()
// writes done by background thread
// it has some overhead, so could be slower
  .asyncWriteEnable()
// increase cache size if you have free heap
// default value is 32K
  .cacheSize(1000000)
  .make();
```

Other DBMaker options

```
// encrypt data with password
  .encryptionEnable(“password”)
// use fast compression
  .compressionEnable()
// enables CRC32 checksum
// to protect from data corruption
  .checksumEnable()
```

Cache options

```
// It caches deserialized objects on heap.
// Default cache size is 32,000, increase it
  .cacheSize(1000000)
// enable least-recently-used cache
  .cacheLRUEnable()
// Unbounded soft-reference cache
// use with plenty of free heap
  .cacheSoftRefEnable()// Hard ref, use if
heap is larger then store
  .cacheHardRefEnable()
```

Concurrent transactions

```
// By default there is single-global
// transaction per store.
// This enables proper transactions
// with full serializable isolation
TxMaker txMaker = DBMaker
        .newFileDB(file)
        .makeTxMaker();

// open two transactions, with single map
// both can only see their own changes
DB tx1 = txMaker.makeTx();
Map map1 = tx1.getTreeMap("map");
DB tx2 = txMaker.makeTx();
Map map2 = tx2.getTreeMap("map");

//commit and close
tx1.commit()
tx2.commit()
txMaker.close()
```

Snapshots

```
// lighter way to get consistent data view
DB  db = DBMaker
        .newFileDB(file)
        .snapshotEnable()
        .make()
Map map = db.getHashMap(“map”);
map.put(1,2);
DB snap = db.snapshot();

Map mapOld = snap.getHashMap(“map”);
map.put(3,4);   //mapOld still has only 1,2
snap.close();    //release resources
```

```
// Third way to ensure consistency is
// Compare and Swap operation. MapDB
// has ConcurrentMap and atomic variables.
```

MapDB Cheat Sheet

www.mapdb.org

Maps and Sets

```
// Shows how to get all available collections
DB db = DBMaker
        .newMemoryDirectDB();
        .make();

// BTreeMap is good for small sorted keys
ConcurrentNavigableMap treeMap =
        db.getTreeMap(“treeMap”);

// HashMap (aka HTreeMap) is good for
// larger keys and as a cache
ConcurrentMap hashMap =
        db.getHashMap(“hashMap”);

// there is also TreeSet and HashSet
SortedSet treeSet = db.getTreeSet(“ts”);
Set  hashSet = db.getHashSet(“hashSet”);
```

Queues

```
// 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”);
```

Atomic records

```
// atomically updated records stored in DB
// Useful for example for sequential IDs.
// there is Long,  Integer, String
// and general atomic variable
Atomic.Long q =db.getAtomicLong(“long”);
q.set(1999);
long id = q.incremendAndGet();
```

Configuring maps

```
// create map optimized for large values
Map<String,String> m =
  db.createTreeMap(“treeMap”);

//serializers are critical for performance
  .keySerializer(BTreeKeySerializer.STRING)
// compress large ASCII string values
  .valueSerializer(
        new Serializer.CompressionWrapper(
                Serializer.STRING_ASCII))
// and store values outside of BTree nodes
  .valuesOutsideNodesEnable()
// enable size counter
  .counterEnable()
// make BTree nodes larger
  .nodeSize(120)
// and finally create map
  .makeOrGet();
```

Secondary indexes

```
// create secondary value (1:1 relation)
// secondary map gets auto updated
Map<ID, Person> persons
Map<ID, Branch> branches
Bind.secondaryValue(persons,branches,
  (person)-> person.getBranch()));

// create secondary key (index) for age(N:1)
SortedSet<Fun.Tuple2<Age,ID>> ages
Bind.secondaryKey(persons, ages,
 (person)-> person.getAge());

// get all persons of age 32
for(ID id: Fun.filter(ages, 32)){
  Person p = persons.get(id)
}
```

HTreeMap as a cache

```
// Entries are removed if map is too large

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

// On-disk cache in temp folder
// with max size 128GB or 1M entries

DB db = DBMaker
        .newTempFileDB()
        .transactionDisable()
        .closeOnJvmShutdown()
        .deleteFilesAfterClose()
        .make()
Map cache = db
        .createHashMap("cache")
        .expireStoreSize(128)  // GB
        .expireMaxSize(1000000)
        .make()
```

Data Pump for faster import

```
// Data Pump creates TreeMap and TreeSet
// in streaming fashion. Import time is linear
// to number of entries.

Iterator iter = … iterate over keys..

Map<K,V> m =  db.createTreeMap("map")
 .pumpSource(iter, (key)-> key.getValue())
 .pumpIgnoreDuplicates()
 .pumpPresort(1000000)
 .make()
```