NTNU – Trondheim
Norwegian University of
Science and Technology

# Evaluation of High Performance Key-Value Stores

**Christian Forfang**

Master of Science in Informatics
Submission date: June 2014
Supervisor: Svein Erik Bratsberg, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

# Summary

Key-value stores have a long history of providing users with a simple to use, yet powerful, interface to durable data storage. Useful for various purposes in their own right, they are also often used as storage layers in more complicated systems—such as fully featured database management systems. For these reasons, it is educational to investigate how such systems are actually architectured and implemented.

This thesis looks in detail at two recently developed embedded key-value stores, Google LevelDB and Symas Lightning Memory-Mapped Database (LMDB). Both systems' overarching architecture and lower-level implementation details are considered to provide a thorough description of how they work.

While exposing largely the same set of features, the systems are shown to take vastly different approaches to data storage. These techniques are described, and put in context with regards to existing ideas from the database management systems field. It is also shown how the designs impact performance in the face of different types of workloads.

# Sammendrag

Lagringsystemer med en ”nkkel-verdi” datamodell har en lang historie av  tilby brukere et brukervennlig, men samtidig kraftig og fleksibelt, grensesnitt for langsiktig og sikker datalarging. I tillegg til  vre nyttig i seg selv er slike systemer ogs ofte brukt som lagringskomponent i strre databasesystemer. Med bakgrunn i dette, er det intressant  underske hvordan et slikt system faktisk er utformet og implementert.

Denne rapporten ser i detalj p to slike nkkel-verdi bibliotek, Google LevelDB og Symas Lightning Memory-Mapped Database (LMDB). Begge systemenes utforming og implementasjon er sett p og beskrevet, slik at en grundig forstelse av hvordan de fungerer kan oppns.

Til tross for at de tilbyr mange av de samme funksjonene, er deres lsningener for datalagring veldig forskjellige. Disse lsningene er beskrevet, og satt i en bredere kontekst. Det demonstreres ogs hvordan teknikkene legger grunnlag for varierende ytelses-karakteristikk i forskjellige bruks-situasjoner.

# Preface

This thesis was written as part of a 2-year Master's degree programme in Informatics at the Norwegian University of Science and Technology (NTNU), in Trondheim, Norway.

The reader is assumed to be familiar with general database management system terminology, concepts, and techniques (such as ACID, B+trees, and write-ahead logging), as well as the C and C++ programming languages. The text, unless otherwise specified, assumes the underlying storage medium used is a hard disk drive (HDD), and not flash memory or similar.

# Table of Contents

# List of Figures

Chapter 1

# Problem Definition and Background

## 1.1 Introduction

The *dictionary problem* is a classic problem in computer science. It involves creating a data structure to store arbitrary values, where each value is associated with a lookup key. Basic operations are then to *find*, *insert*, and *delete* values associated with a given key. Such a structure is also referred to as a *dictionary*, *associative array*, *map*, *symbol table*, or *property list*.

The challenge is to provide these operations in an efficient manner. Research in this area has a long history, and innumerable solutions have been proposed. These often use various kinds of trees,(Anderson, 1989) or some type of hashing.(Knuth, 1973)

A *key-value store* is a data storage solution implementing a dictionary data model. One of the early examples of such a data store is *dbm*. Created by Ken Thompson and released in 1979, it was implemented using on-disk hash tables. Being a program library, users could link their own programs to it and have *dbm* handle the—otherwise somewhat complicated—issue of durable and efficient disk-based storage. Supported operations, in addition to insertions of key-value pairs, included search and retrieval of data through its key, deletion of a key along with its data, and to sequentially iterate over all stored key-value pairs (in an order defined by the hashing function). Since its release, many derivatives have been created implementing a similar API.(Poznyakoff, 2011)

A later key-value store is *Berkeley DB*. Released in 1991, it began life as a new implementation of a hash-based access method, intended to replace various existing *dbm* implementations. Later it was extended to also include a B+tree access method, and had additional features added in addition to the minimal *dbm* API—for example extensions to handle concurrent access, and transaction support. To emphasise its use as a program library (as with *dbm*), it is billed as an *embedded* data store. This is to differentiate it from database *server* processes—though such a system could easily be built on top of for example Berkeley DB.(Michael A. Olson, 1999) More recently, the NoSQL movement has popularized the term *key-value store* to refer to a particular type of database server exporting a key-value data model—often with additional distributed features such as replication—further necessitating this distinction.(Cattell, 2011)

The problem of efficiently implementing an embedded key-value store is more specialised than the basic dictionary problem. Users often want the data to be stored to a more durable storage medium than volatile memory, which has widely different performance characteristics. Much focus has therefore been on creating and evaluating different disk-based access methods and indexing structures, such as B+trees and different kinds of hashing, when developing new systems.

Below, the idea of an embedded key-value store is further explained. Then some of the tradeoffs which can be made when implementing such storage systems is covered. Finally, motivation for writing the thesis, and the contents of further chapters, are described.

## 1.2 Embedded Key-Value Stores

The features provided by an embedded key-value store can vary significantly between different systems. This section will describe some of the more popular ones, and present some examples of how such storage solutions are used in practice.

**Get/Put/Delete API** Being able to `Get` the stored value of a given key, `Put` a key-value pair into storage, and `Delete` a stored key-value pair is the basic capability all key-value stores will provide.

**In-memory only data storage** If it is not necessary to store the data durably, a system could store the data entirely in memory. This would greatly speed up access times, but limit the total data amount which can be stored—the exact amount depending on the available memory address space.

**Transactions** Transactions give users the ability to perform a sequence of changes which succeeds or fails as a unit. A classic example of why this is desirable is the transfer of money from one bank account to another. In such a case, a user would like to have a guarantee that, if the withdrawal-operation succeeds, the deposit-operation will complete as well—otherwise money could be lost in the event of a crash.

**Ordered iteration** A user might find it desirable to iterate over all data in the database ordered by key-value. This may or may not be easily possible depending on the database implementation.

**Snapshots** It is often desirable to be able to have access to a *snapshot* of the database. This represents the state of the database as it was at the time of the snapshot's creation, and enables users to access all database-contents as they were at that time—even if changes has been made since.

**Concurrency** Multi-threaded or multi-process concurrency enables users to access the database from multiple threads or processes simultaneously, without external synchronization.

### Use Cases

The general problem an embedded key-value store addresses is that of *fast and reliable local key-value storage*. Here, *local* storage implies it happens on the computer the software is running on. To illustrate where this might be useful, it is common for many types of applications to want to persistently store some kind of information between executions—and to do this locally, without being reliant on the state of an external server. This can be simple things like configuration state or any other kind of application-specific data. Embedded key-value stores can fulfill this need by providing a very simple, but also flexible, data model, allowing the application to focus on more domain-specific tasks unrelated to durable data storage.

Embedded key-value stores are also frequently used as a *storage layer*[1] for more complex data-storage solutions. Here it handles the more limited scope of durable data-storage only, allowing other higher-level database components to provide whichever features the database need to provide. Such features can for example be support for an SQL interface (or some other data-model), ACID transactions, distributed sharding and/or replication, or just a simple network interface.(Burd, 2011)

With a many different kinds of embedded key-value stores available to choose from, often with the same basic set of features, it becomes necessary to look deeper to get a sense of their relative pros and cons.

## 1.3 Data Storage Performance Tradeoffs

With any type of dictionary, the implementation of it will determine the efficiency and performance of a given operation. Considering only *inserts* and *lookups*, the structure used

[1] Or *storage engine*.

will necessarily introduce a *tradeoff* between the performance of these two operations. (Walsh, 2011)

Two illustrate this, consider a system where raw *insert* performance is the only consideration. The best one can do in this case is to fully utilize the bandwidth of the underlying storage device, for example by logging all data to disk with no additional processing. This will maximize insert-performance, but make lookups extremely slow as each lookup will, on average, require half the written data to be examined. Depending on how the system is used, this may or may not be acceptable from a performance standpoint. If lookups are sufficiently rare, or if unordered iteration over all the stored data is the only additional access method needed, the superior write performance might offset this disadvantage. If not, a user might find the lookup performance unacceptably slow.

The opposite extreme is a *read optimized* system, where lookup performance is maximized. What an optimal system is, in this case, will depend heavily on the access pattern of the lookups. A system which could perform well in most cases is a hash-table where each insert causes the system to generate a new perfect hash-function—and re-hashes all existing items in the table. As the number of items grow this becomes an extremely expensive operation, but lookup performance will be excellent. Another alternative could be to store data multiple times in different formats so that the optimal format is always available for a given access scenario.

A more general system will want to hit some kind of tradeoff between these extremes; a system where both insertions and lookups are reasonable efficient. To achieve this, both the system architecture (including the access methods it uses), and implementation, need to be carefully considered.

## 1.4 Motivation and Thesis Contents

When reading published papers or other descriptions of how a given system works, it is rarely possible to attain anything but a very high-level architectural understanding of it. This is certainly useful, but such a description will necessarily simplify or gloss over the more gritty details of *how* it works—relating to more low-level implementation details—if for no other reasons that creating such a description can be very time consuming, while comparatively providing little value to the average reader. However, to fully understand how a system actually achieved what it does, it is often necessary to look at it at this level of detail.

For this reason, studying the actual source-code of software can be very useful, as it fully defines how a system works. For someone unfamiliar with any type of software—be it data-storage-solutions, operating systems, or video games engines—studying the source-code of one such system is likely to give the reader great insights into both how both that specific system works, as well as how other related systems are likely to function. This goes for both a high-level architectural understanding, as well as more implementation-focused details.

Motivated by the above reasoning, the rest of this thesis will look at two existing embedded key-value stores, Google LevelDB and Symas LMDB. These are both relatively recent systems, initially released in 2011 and continuously update since. The aim is to provide a sense for how such systems are actually architectured and implemented, as well as to compare and contrast the two implementations in terms of the architecture used, features provided, and performance in different scenarios. Having this level of knowledge is both useful in itself, also helpful to be able to select between these, and similar, systems when faced with a given usage scenario.

Chapter 2

# Case Study: Google LevelDB

## 2.1 Introduction

LevelDB describes itself as a *fast key-value storage engine written at Google that provides an ordered mapping from string keys to string values*. It is created primarily by Jeff Dean and Sanjay Ghemawat.(Jeff Dean, 2011) Written in C++, it was first released publicly in 2011 and has since seen multiple revisions, with the most recent version (1.16) being released in February 2014. It is licensed under a BSD-style open source license and has, in addition to seeing much use on its own, also served as the base for multiple other database projects.(LevelDB, 2014)

As an embedded key-value store it can serve multiple purposes. Among these, Google highlights using it to *"store a cache of recently accessed web pages [in a web browser]"*, *"store the list of installed packages and package dependencies [in an operating system]"*, and *"store user preference settings [in an application]"*. It is additionally designed to be *"useful as a building block for higher-level storage systems"*.(Jeff Dean, 2011)

Specific example of its usage include inside the Google Chrome web browser, where it is used to implement the IndexedDB[1] HTML5 API, in the Riak database where it is available as a one of its storage backends,(Basho, 2014b), in Apache ActiveMQ as a persistent data store,(Apache, 2014) and in the Bitcoin (and derivatives) software for storing transaction and block data.(Bitcoin, 2013)

### Features

The following are some of the features of LevelDB, as listed on its project website:(LevelDB, 2014)

[1] http://www.w3.org/TR/IndexedDB/

- Keys and values can be arbitrary byte arrays.
- Data is stored sorted by key.
- Callers can provide a custom comparison function to override the sort order.
- The basic operations are `Put(key,value)`, `Get(key)`, and `Delete(key)`.
- Multiple changes can be made in one atomic batch.
- Users can create a transient snapshot to get a consistent view of data.
- Forward and backward iteration over the data is supported.
- Data is automatically compressed using the Snappy compression library.[2]
- External activity (file system operations and similar) is relayed through a virtual interface so users can customize the operating system interactions.
- Detailed documentation about how to use the library is included with the source code.

## Chapter contents

The sections in this chapter will contain the following: first LevelDB's API and how is used in a C++ application is presented. Then its architecture and implementation is covered in some detail, followed by a brief look at some projects of which LevelDB has served as a base. The final section will discuss the various approaches used by LevelDB, and put these in a broader context.

[2]https://code.google.com/p/snappy/

## 2.2 Usage Example

This section will describe the API exported by LevelDB, and show some examples of its usage.

LevelDB presents a reasonably small interface to clients: besides the basic `Get`, `Put`, and `Delete` operations, it includes methods to create and release a snapshot of the current state, a method to create a new iterator over the current state, and methods to query data storage statistics. Below is the complete public C++ interface exposed by LevelDB:

```
static Status Open(const Options& options, const std::string& name,  DB** dbptr);

virtual Status Get(const ReadOptions& options, const Slice& key,
                   std::string* value) = 0;

virtual Status Put(const WriteOptions& options, const Slice& key,
                   const Slice& value) = 0;

virtual Status Delete(const WriteOptions& options, const Slice& key) = 0;

virtual Status Write(const WriteOptions& options, WriteBatch* updates) = 0;

virtual Iterator* NewIterator(const ReadOptions& options) = 0;

virtual const Snapshot* GetSnapshot() = 0;

virtual void ReleaseSnapshot(const Snapshot* snapshot) = 0;

virtual bool GetProperty(const Slice& property, std::string* value) = 0;

virtual void GetApproximateSizes(const Range* range, int n, uint64_t* sizes) = 0;

virtual void CompactRange(const Slice* begin, const Slice* end) = 0;
```

The basic usage of LevelDB is relatively straightforward: first a database-object is created. This opens an existing database (and performs recovery if needed), or creates a new one.

```
leveldb::DB* db;
leveldb::Options options;
options.create_if_missing = true;

leveldb::Status status = leveldb::DB::Open(options, "testdb", &db);
```

The `leveldb::Status` class is used throughout LevelDB to return information about the success or failure of an operation. It can be used as follows:

```
if (!status.ok())
    std::cerr << status.ToString() << std::endl;
```

To close a database, one simply `delete`s the database object:

```
delete db;
```

Reads and writes can be performed using the database ojbect's `Get`, `Put` and `Delete` methods:

```
std::string key1 = "...";
std::string key2 = "...";
std::string value;

leveldb::Status s = db->Get(leveldb::ReadOptions(), key1, &value);

if (s.ok())
   s = db->Put(leveldb::WriteOptions(), key2, value);

if (s.ok())
   s = db->Delete(leveldb::WriteOptions(), key1);
```

. . . and iteration over all key-value pairs can be done through an iterator:

```
leveldb::Iterator* it = db->NewIterator(leveldb::ReadOptions());

for (it->SeekToFirst(); it->Valid(); it->Next()) {
   cout << it->key().ToString() << ": "  << it->value().ToString() << endl;
}

delete it;
```

Finally, creation and use of a snapshot looks like the following:

```
leveldb::ReadOptions options;
options.snapshot = db->GetSnapshot();

//  ... apply some updates to db ...

leveldb::Iterator* iter = db->NewIterator(options);
//  ... read using iter to view the state when the snapshot was created ...
delete iter;

db->ReleaseSnapshot(options.snapshot);
```

From the interface, it is worth noting how the key and value-arguments are of type `leveldb::Slice`—except for `Get` which takes a pointer to a `std::string` for its value argument. It is crucial to understand what a `Slice` represents as it is also used extensively internally in LevelDB: it is simply a (non-owning) pointer to a byte-array stored externally, along with the size of this data.

The `Slice` class contains C++ constructors which can take references to a `std::string` or `char*` and then stores a pointer to this data along with the size (extracted using `std::string::size()` or `strlen(const char*)`. One can also construct a `Slice` by giving it this information directly, for example if it is not contained in a `std::string`, or is a `char*` containing `\0`' bytes—meaning `strlen` will not return the correct size. Since the `Slice` does not own the data it points to, it important the data is kept alive for the lifetime of the `Slice`.

The `Get` method takes a pointer to a `std::string` which will be "filled out" by LevelDB (by copying from internal buffers) if the value for the key in question is found. This string thus takes ownership of the returned data and makes sure the memory is freed on its destruction.

The `leveldb::WriteOptions` argument is used to convey whether or not a change should be flushed from the operating system buffer cache, through its `sync` flag. For reads, the `leveldb::ReadOptions` argument conveys options like whether or not file-checksums should be verified, if the data read should be excluded from internal caching, and information about which snapshot of the database the read should take place in (if any).

Calls to the database-object can be performed from multiple threads simultaneously without external locking. Additional details about concurrency in LevelDB can be found in section 2.4.6.

Another noteworthy operation is the ability to perform a set of updates atomically. This is covered with examples in section 2.4.10. Other operations, including setup of internal comparators for keys (to define the ordering among them), and more advanced options like cache- and filter-setup, is not covered here. The interested reader should instead check out LevelDB's own documentation.(LevelDB, 2012)

## 2.3 Architecture Overview

### 2.3.1 Introduction

This section will cover LevelDB's high-level architecture and its major components. This should give the reader a general understanding of how LevelDB works, and lays the foundation for later talking about its actual implementation.

First, the central concepts of memtables and sstables are introduced. Then, the levels structure—from which LevelDB takes its name—and the necessary compaction-process to maintain it are described. Next, the path a given write- or read-operation takes through the systems is covered. The remainder of the section will then look at the issues of providing snapshot-support and iterators for the database-contents, how durability and recovery is handled, and finally a description of how LevelDB does in-memory caching of data.

### 2.3.2 Memtable and Sstables

LevelDB uses two kinds of structures to store data internally. These are called "memtables" and "sorted string tables" (sstables) respectively. All data in LevelDB is contained in one of these two structures.

#### Memtable

As the name suggests, a memtable is a completely in-memory data structure. It is used to hold a set of entries in sorted order by the entries' keys. Entries can either be key-value pairs, or a deletion marker for the key. While holding this data, the memtable can be queried to return entries by key, asked to provide an iterator over all entries it contains, and have new entries inserted into it. Since the memtable is an in-memory data structure, these operations can all be expected to be fast as no disk IO is involved.[3] A given memtable is generally kept small, at maximum a few megabytes in size. Once it reaches a certain size it is frozen (no more entries are inserted), replaced, and its contents eventually moved to disk.

#### Sstable

An sstable—or "sorted string table"—is a custom file format which stores a sequence of entries sorted by key. Similar to the memtable, each entry can either be a key-value pair or a deletion marker for that key. LevelDB tries to keep each sstable sized to around 2 MB, but the exact size depends on the entries contained in it and any compression applied.

[3] This is a simplification: for durability-reasons an on-disk log of inserts to the memtable is kept—but this is not handled by the memtable itself.

The sstable is structured so that it is possible to tell approximately where in the file a given entry must be located, if it exists. This is done by first storing all entries in sorted order—hence the name "sorted string table"—and then including an index at the end of the file. This is not a complete index of all entries, but rather the location of all *"blocks"* in the file—each of which contain a number of entries—and the first and last (equivalently, smallest and largest) entry stored by each block. By having this index available, the position of the block where a given entry *must* be located (if it exists in the file) can be calculated, and then the block retrieved with a single disk-seek. This makes the sstables very efficient for lookups.

In addition to blocks containing entry-data, the sstable format also allows for blocks with auxiliary information. This could for example be a bloom-filter over all entries contained in the file.[4]

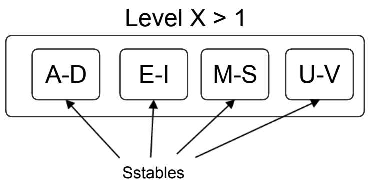

**Figure 2.1:** A single level consisting of sstables with non-overlapping key ranges.

## 2.3.3 Levels and Compactions

The sstable files used to hold disk-resident data in LevelDB are organized into a sequence of levels. The levels number from level-0 to a maximum level, by default 7. Except for the special level-0, all sstables in a given level have distinct non-overlapping key ranges (see figure 2.1). This is done so that when an entry is searched for, only *one* sstable in each level can possibly contain that key—namely the sstable which has a range overlapping the searched-for key (if one exist). By maintaining this invariant, lookups can be done very efficiently: first the one sstable in the level where a sought-after entry must be located is identified, then the sstable block (potentially) containing the entry itself is directly retrieved with a single disk-seek after consulting the sstable index for its location. (This assumes the sstable index is available, else an additional seek is required to first fetch the index.) See figure 2.2 for an illustration of the levels structure.

As opposed to the larger-numbered levels, level-0 does not necessarily contain sstables with overlapping key-ranges. This is done so that once a memtable fills up and is frozen, is can immediately be converted to an sstable and written to level-0. Had level-0 required non-overlapping key-ranges, additional work would have to be performed to merge in the memtable's content. As a consequence of this, a key-lookup in level-0 need to look at *all* files in the level—meaning the read-performance decrease as the number of files increase.

[4] A bloom-filter is a data structure which can quickly and efficiently determine if a given element can, or can not, be a member of a set. By consulting a stored bloom-filter first, some lookups (and subsequent IO operations) can be avoided if the bloom-filter determines a search-for entry is guaranteed to *not* exist in the file.

To counteract this, when the number of files in the level-0 reaches a threshold value LevelDB *merges* some of the files in level-0 together with overlapping files in the next level —level-1—to produce a new sequence of level-1 files. This process lowers the number of level-0 files while still maintaining the non-overlapping key-range invariant of level-1. LevelDB refers to this merge as a *compaction*. See figure 2.3 for an illustration. When merging the contents of the sstables, LevelDB also identifies instances of entries with the same key and removes all but the most recent version—with some exceptions to support snapshots—and also removes tombstone-entries if it can guarantee no entries with that key exists in higher-numbered levels.

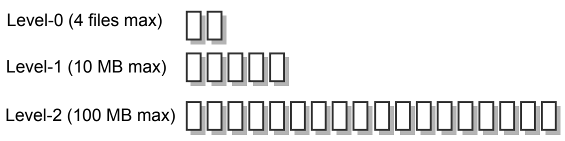

**Figure 2.2:** Illustration of three levels in LevelDB. Each square is an sstable.

For levels other than level-0 (except the final level), files are compacted to the next level once the combined size of all files in the level reaches a maximum.[5] This size is $10^L$ MB, where L is the level-number. (In other words, 10 MB for level-1, 100 MB for level-2, and so on.) This has the effect of gradually migrating updates from *lower* to *higher* levels, while only using bulk read and writes.

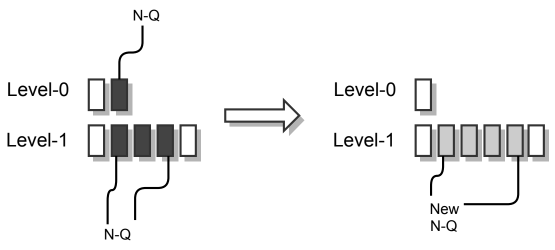

**Figure 2.3:** Illustration of a level-0 to level-1 compaction. All files in level-1 which overlap the level-0-file's range is merged with the level-0 file to form a new sequence of level-1 files.

To get a sense for why this is done, imagine a scenario where only level-0 to level-1 compactions are done. Over time, the number of level-1 will then continue to grow as level-0 sstables are compacted, but because of the non-overlapping range-invariant read-times will not increase—only a single sstable can possibly contain the entry for a given key. However, once a new level-0 sstable is to be compacted, the number of level-1 sstables it overlaps is likely to go up with the actual number of level-1 files. To construct the new sequence of level-1 files, *all* the data in these overlapped files will have to be rewritten (to maintain the sorted order)—a major overhead in terms of disk IO—while only around 2 MB of new data is actually introduced (the contents of the level-0 sstable). By instead

[5]There are also some other triggers which can cause such a compaction.

maintaining a greater number of levels of exponentially increasing size, this overhead is kept to a minimum.

### 2.3.4 Write Path

Both `Put` and `Delete` operations are handled by LevelDB by adding *new* entries to the database. This is done instead of changing or removing existing entries in order to provide snapshot support. This section will look at what LevelDB does when such a write (addition of a list of entries) is requested by a client.

As illustrated by figure 2.4, all entries are initially added to a memtable—of which there is only one active at a given time (the *current* memtable). For durability reasons, all writes are also logged to disk so that the contents can be recovered should the system crash (which would cause the loss of the memtable as it is located in volatile memory). Once this is done, the write is considered finished and control returns to the client requesting the write.

Before being added to the memtable all entries are tagged by LevelDB with a monotonically increasing sequence number. This is the mechanism used by LevelDB to globally order entries, and enables LevelDB to disambiguate two entries with the same key. (The entry with the higher sequence number will be the newer version of the two.)

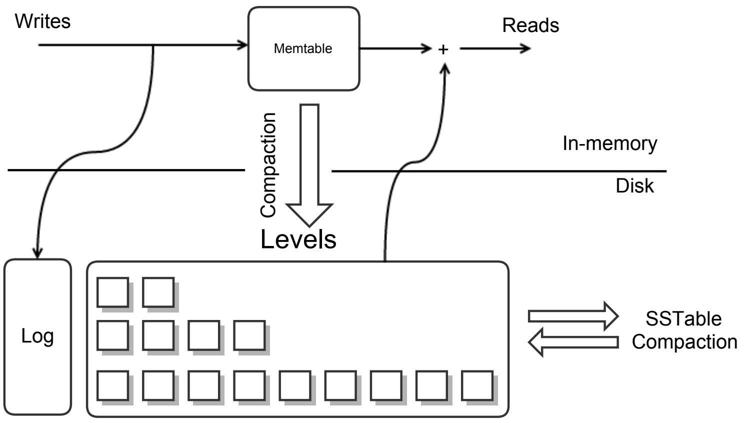

**Figure 2.4:** Illustration of data-movement in LevelDB. Writes only update the memtable, while reads combine data from both the memtable and the on-disk levels (containing sstables). Once a memtable fills up it is compacted to disk, where it is stored as an sstable. Compactions are also done between levels as they reach their size thresholds.

As writes are done by inserting them in the memtable, the memtable eventually reaches a configurable maximum size threshold. This triggers a *compaction* to convert it to a level-0 sstable, illustrated by the arrow pointing from the memtable to the levels structure in figure 2.4. (Before this is done, a new memtable is set up to serve future write requests.) The memtable scheduled for compaction is referred to as an “immutable” memtable, and is kept available for read operations until its contents are finalized to disk. Once a memtable, and consequently the data it contains, has been added to disk as part of a level, the immutable memtable can have its memory freed. The log file associated with the memtable can then also be discarded.

### 2.3.5 Read Path

When a `Get` (or lookup/read) operation is requested, LevelDB has to look at both the memtable and on-disk sstable-contents to try to find the entry for the requested key. This can be done in order: first by checking the memtable, then possibly the immutable memtable (if it exists), before finally going to disk where it checks each level in turn.

When going in this order, the search can stop the moment an entry with the searched-for key is found. This is because any entries with the same key found in any of the subsequent locations will necessarily be *older* (have a lower sequence number) than the found entry. This gives an advantage to lookups of recently added entries, as they will be found quickly—possibly even in the in-memory sstable, not requiring any disk IO at all. By checking if the found entry is a key-value pair or a tombstone, LevelDB is able to either return the value or notify the caller that the entry has been deleted.

When the levels structure is considered, the read process looks at each level in turn. For level-0, this means potentially looking at *all* sstables in the level—assuming they have a range which implies it could contain the searched-for key—at worst requiring two disk-seeks to check each one. For levels greater than 0, only one sstable can potentially contain the entry so only two disk-seeks, at worst, is need to check each level. As already mentioned, once an entry is found subsequent levels does not need to be searched.

Because of sstables are read-only, it is safe for multiple threads to read them simultaneously in a lock-free manner. Similarly, the memtable is constructed so that it can be traversed without acquiring any locks. More details on how this is done can be found in the implementation section.

### 2.3.6 Snapshots and Iterators

LevelDB supports the notion of a “snapshot” of the database: a reference to the contents (state) of the database as it were at a given point in time. A snapshot can be created explicitly through the `GetSnapshot()`-method; this returns a handle which can then be passed to the database `Get`-method through its `ReadOptions` parameter. The lookup operation will then be performed on the database state as it were, content wise, at the time of snapshot-creation, disregarding any future changes.

Iterators, created through calls to `NewIterator(const ReadOptions& options)`, can be used to iterate over the contents of the database as it were when the iterator was created, in sorted key order. This feature can additionally be combined with snapshots so that the iterator is instead over the contents of a given database snapshot.

How LevelDB's architecture allows for these two features is the subject of this section.

### Snapshots

For snapshots, the basic requirement is that all entries at the time of the snapshot's creation continues to be available, and that new entries, and updates to existing entries, can be identified as such so that they are not returned. Because LevelDB stores a monotonically increasing sequence number along with all entries, entries newer than a given snapshot can easily be identified. As long as sufficiently old entries are kept alive, LevelDB can then ignore new versions and instead return the newest version still older than the point in time the snapshot was made. In other words, the "current" entry for a snapshot is the entry whose sequence number is smaller than or equal to the snapshot's number while still larger than any other entries with that key.[6]

By tracking which snapshots are live at all times, LevelDB can then make sure to keep alive any entries which, even though they have been superseeded by a new version, might still be current for any live snapshot. In contrast, if no snapshots are live, a new version of any key means the old version could be immediately deleted as any lookups would only be interested in this newer version.

### Iterators

To provide iterators, LevelDB needs a way to combine the different data-sources (the memtables and the sstables in all levels) to provide a global, unified, iterable, and sorted view of the database. Because levels greater than 0, and all memtables, are already sorted in key order, this is relatively straightforward: LevelDB simply merges iterators provided by the memtables (of which there can be a maximum of 2—the current and immutable memtable), levels 1 to max (of which there can be a maximum of max-levels minus 1), and all sstables in level-0 (of which LevelDB tries to keep to a number no more than 4). By comparing the keys returned by each of these iterators and controlling the iterators' progression, a "merging" iterator can return the next value in global order. This also makes it easy to iterate over a snapshot: as with snapshot lookups, the merging iterator can ignore values with too large sequence numbers, and once a value for a given key is found, skip over old versions of that key.

The main challenge to this scheme is how to handle changes to the database made after the iterator is created—in other words changes to the memtable and changes to the levels introduced by compactions. For the memtable, it is sufficient to skip over any new entries having been added since the iterator was created (as identified by their sequence numbers). By then also making sure the memtable itself is not deleted (as it otherwise would be after being compacted to disk), the iterator can keep using it for as long as it needs. For levels, LevelDB opts to *not* try to handle this in any complex way—instead it simply notes the current state of all levels at the time of iterator creation, and then guarantees the sstables

[6]In other words, it is the most recent version created before the snapshot's logical place in time.

in question will not be deleted (as they otherwise would be after a successful compaction) until the iterator is. This way, the merging iterator can proceed knowing no changes to the database (in terms of compactions, or replacements, of memtables and sstables) can corrupt any of its sub-iterators. Since the iterator is not interested in any future updates to the database anyway (it works on either an explicit snapshot or an an implicit snapshot taken at the time of its creation), this is not an issue. A drawback to this approach is that both disk-space- and memory-usage potentially increases while an iterator is active. This is because neither the memtables, nor any sstables alive during its creation, can be deleted before the iterator is, in order to provide the consistent view it requires.

Noting how snapshots work, that is by not deleting certain entries during compactions but otherwise proceeding "as normal", its worth questioning if the approach used for iterators could be used for snapshots as well. In other words, having the creation of a snapshot also simply store the current set of sstables and memtables and then instruct LevelDB to keep these alive until the snapshot is deleted. While the answer is that it probably could, the heavyweight nature of this operation is likely the reason snapshots are handled differently. The approach used for snapshots does not come without its own set of drawbacks, but this is only in terms of disk space—not memory-usage—as old versions of entries are alive. Iterators additionally increases memory usage by forcing memtables to stay alive. (Each iterator can at worst force two memtables to not be deleted for the iterator's lifetime.) With memory likely being the more precious resource, it seems fair to assume this is the reason LevelDB does not unify the two techniques.

### 2.3.7 Recovery and Durability

To provide durability in the face of system crashes, a database has to make sure changes to it are applied in such a manner that they can be recovered should the system fail. In LevelDB there are two such causes of change: writes (addition of entries, including tombstones) and compactions. It is thus necessary that both of these are done in such a way that data-loss is not possible should the system crash at an inopportune time.

As illustrated by figure 2.4 on page 15, LevelDB ensured the memtable contents can be recovered by recording all updates to it in a log. Because this greatly increases the overhead of mutating the memtable, it is optionally possible to disable this behaviour—for increased performance—if data loss in the event of a crash is acceptable. On restart after a crash, LevelDB will locate all log-files and recover the memtables they represent. By logging entries in their entirety, as LevelDB does, this is a trivial operation, and by marking all log-entries with CRC checksums, half-written (incomplete) log-entries at the end of the file can be ignored.

That covers one part of the system, but still leaves the changes introduced through compactions. This includes both "minor" compactions (memtable to sstable) and "major" compactions (merging of sstables across levels). A key insight in this instance is that sstables are immutable. This allows the current database state to be represented completely by a list of sstable-files at each level together with the memtable log files (of which there can be two—the current memtable's log, and the immutable memtable's log). Because the compaction process only creates *new* files, the current state of the database will never be corrupted by it—only by *deleting* in-use files can one run into problems. The compaction process is thus free to construct a new database state in the background in whatever manner it wants. Once this new state is created, likely consisting of many of the same files as the previous state but with some sstables merged (compacted) to new files, this new state can atomically be swapped in to become the current state. By logging this process, and not deleting any of the old state's files before this new state and its files are deemed durable (through being written to disk), the current state can always be recovered should the construction of the new state fail at any point. The current active memtable (rather, its log) is adopted by the new state so that no updates are lost when moving to a new (post-compaction) state.

To illustrate this process in the case of immutable-memtable (minor) compaction, consider the following:

1. The current state of the database is this: a current and immutable memtable exist—their contents are reflected by log files—and a list of sstables at each level is somehow stored. The compaction-process wants to compact the immutable memtable to an sstable. It creates a copy of the current state which it can freely modify.

2. The compaction-process creates an sstable from the immutable memtable, and places it in the appropriate level (the list of sstables at each level is updated). It registers that the immutable memtable's log file is not part of the new state.

3. The new state is this (compare with point 1): a current memtable represented by its log file, and a list of sstables at each level.

4. After a representation of the new state is durably written to disk, so that it can be recovered in the event of a crash, the current state can safely be replaced by this new state.

Note how no data is lost in this transition:[7] the current memtable is not changed (it is shared between the states), and the immutable memtable's content is not lost but rather converted to an sstable where it can still be found. Since the new state does not contain an immutable memtable the current memtable can later be replaced (and become an immutable memtable awaiting compaction) once it is full. "Major" compaction (sstable to sstable) progress in the same way, but only the list of files in each level is changed.

The new state is subsequently used to serve new requests, but the old state might still be kept around—for instance if it is used by a live iterator. Once it is no longer needed, any files it references which is not part of any newer states can be deleted to free up disk space.

### 2.3.8 Table and Block Caching

As sstables are immutable, it is easy for LevelDB to cache their contents in memory without having deal with a complex buffer management scheme. The focus can be purely

[7] No *needed* data. The compaction process will discard as much unneeded, old, data as possible.

on speeding up reads by caching as much of the database's working-set in memory at all times. To this end, LevelDB includes both a table- and block-cache:

- The table-cache is used to keep the sstables' index-blocks in-memory so that candidate blocks in these sstables can be located directly. By default, LevelDB creates a table-cache with space for 1000 tables. This can hold index-blocks for around 2 GB of on disk data.[8]
- The block cache is used to hold full sstable blocks to avoid reading them from disk on subsequent accesses. The block cache is fully user configurable, but unless specified LevelDB creates an 8 MB cache by default.

More details in how the two caches work can be found in the implementation section below.

[8] Assuming sstable sizes of 2 MB.

## 2.4 Implementation Highlights

### 2.4.1 Introduction

Having looked at the general high-level architecture of LevelDB in the previous section, the focus here will be on the actual implementation of some of the concepts mentioned.

First, the the internal format used by LevelDB for keys is described. Then, the implementation of the memtable is covered, and the *Version* and *VersionSet* classes and their purpose is described. Next, the details of the sstable file format is covered, before the issues of concurrency and how reads and writes are done at the implementation-level is described. Finally, details on the compaction-process, caching, snapshots and iterators, and finally recovery, is discussed.

### 2.4.2 Internal Key Representation

Since there can exist multiple entries for a given key in the database—older and new versions, where both are kept around either awaiting deletion on compaction or for snapshot support—the internal key for an entry in LevelDB includes a sequence number as well as a type-specifier to indicate tombstones. This internal key has the form illustrated by figure 2.5: it consists of the actual user-specified key (an array of bytes) followed by a 64-bits number encoding the sequence number of the key along with its type (stored in the lower 8 bits). The type is used to mark the entry as either a normal key-value pair or a tombstone.

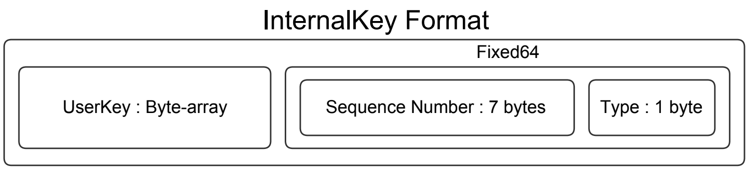

**Figure 2.5:** LevelDB Internal-key format

To order these internal keys, LevelDB uses an comparator implemented as follows:

```
int InternalKeyComparator::Compare(const Slice& akey, const Slice& bkey) const {
  // Order by:
  //    increasing user key (according to user-supplied comparator)
  //    decreasing sequence number
  //    decreasing type (though sequence# should be enough to disambiguate)
  int r = user_comparator_->Compare(ExtractUserKey(akey), ExtractUserKey(bkey));
  if (r == 0) {
    const uint64_t anum = DecodeFixed64(akey.data() + akey.size() - 8);
    const uint64_t bnum = DecodeFixed64(bkey.data() + bkey.size() - 8);
    if (anum > bnum) {
      r = -1;
    } else if (anum < bnum) {
      r = +1;
    }
  }
  return r;
}
```

This orders the keys first by by increasing user key (as specified by a potentially client-provided custom comparator), then decreasing sequence number. The end result is that, when iterating keys in sorted order, the different versions of a given key will appear by decreasing age—newest version first, then progressively older versions. This makes it easy to ignore specific versions of keys, be that older versions (for example when iterating the current state), or newer versions (when doing snapshot-lookups).

The internal key itself does not contain any information on the length of the key. This is instead stored by the structure in which the key is contained, for example a memtable or sstable block.

### 2.4.3 Memtable

The memtable can be considered in relative isolation, as it does not itself interact with any other parts of LevelDB. Below is a look at the interface it exports, as well as how it implements this functionality.

#### Interface

The memtable exports the following public interface:

```
explicit MemTable(const InternalKeyComparator& comparator);

void Ref();

void Unref();

Iterator* NewIterator();

void Add(SequenceNumber seq, ValueType type, const Slice& key, const Slice& value);

bool Get(const LookupKey& key, std::string* value, Status* s);

size_t ApproximateMemoryUsage();
```

The `Ref()`- and `Unref()`-functions are used to implement reference counting. This is used to control the lifetime of the memtable so that it can be kept alive while needed, without regard to why or who keeps it alive. `NewIterator()`, unsurprisingly, returns an iterator over key-value pairs stored by the memtable.

The two primary operations on the memtable are `Add(...)` and `Get(...)`. These are used to respectively insert and performs lookups of key-value entries. There is no deletion-method: any entries added to the memtable are guaranteed to persist. Deletions are instead done by adding deletion-marking tombstones.

To be able to construct the internal database key (section 2.4.2), the memtable's `Add(...)`-method does not only take a user-key-value pair, but also a `SequenceNumber` and a `ValueType`. The `Get(...)` method similarly takes a `LookupKey` as a parameter. This is an object containing a representation of the searched-for key directly comparable to the memtable's own internal representation, as well as internal- and user-key representations.

A request for a key through the `Get(...)` method can return three possible results: value found (in which case it is returned through the `value` parameter), key has been deleted (indicated through the status parameter), or entry not found. The first two cases are indicated by `Get(...)` returning true and the latter by it returning false.

### Implementation

With an understanding of what operations the memtable needs to support, the following covers its implementation.

The private variables of the class are the following:

```
struct KeyComparator {
  const InternalKeyComparator comparator;
  explicit KeyComparator(const InternalKeyComparator& c) : comparator(c) { }
  int operator()(const char* a, const char* b) const;
};
KeyComparator comparator_;

int refs_;
Arena arena_;

typedef SkipList<const char*, KeyComparator> Table;
Table table_;
```

The `KeyComparator` is used to order entries in the memtable: it extracts an internal-key from the memtable's internal representation (see below) and passes it on to the `InternalKeyComparator` assigned to the memtable on its construction.

The `refs_` variable is used to keep the reference count of the memtable, while the `arena_` is the memory-arena used to allocate space for the memtable. This is used, instead of allocating memory through `new` or `malloc` directly, to provide a degree of locality for data in the memtable. Once the memtable is deleted all memory allocated by the arena is automatically freed.

Finally, the implementation contains a *skip-list* of `const char*` entries. This is the central data structure used by the memtable to store the added key-value pairs. It supports both efficient (on average O(log N)) insertions and lookups, and also has the important property that it can support concurrent reads while a writer-thread is changing the list—without the need for locks.[9] This fact is a notable advantage the skip-list has over other candidate data-structures, and a very likely reason for why it was chosen.

[9]Some atomic operations are required however, for instance when updating pointers.

How the skip-list itself works is described in section 2.4.3 below. For now, it is sufficient to consider it a type of linked list.

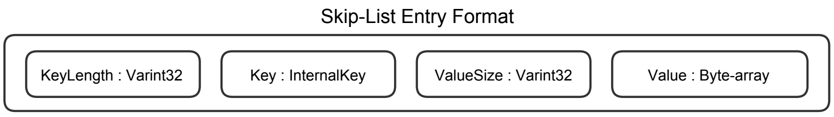

**Figure 2.6:** LevelDB skip-list entry format

The entries in the skip-list contains pointers to a textual representation of a single key-value pair. It has the format depicted in figure 2.6: the key and value is stored sequentially, but prefixed by their length. This representation is created in `Memtable::Add(...)`:

```
 1  void MemTable::Add(SequenceNumber s, ValueType type,
 2                     const Slice& key,
 3                     const Slice& value) {
 4    // Format of an entry is concatenation of:
 5    //  key_size     : varint32 of internal_key.size()
 6    //  key bytes    : char[internal_key.size()]
 7    //  value_size   : varint32 of value.size()
 8    //  value bytes  : char[value.size()]
 9    uint32_t key_size = static_cast<uint32_t>(key.size());
10    uint32_t val_size = static_cast<uint32_t>(value.size());
11    uint32_t internal_key_size = key_size + 8;
12    const size_t encoded_len =
13        VarintLength(internal_key_size) + internal_key_size +
14        VarintLength(val_size) + val_size;
15    char* buf = arena_.Allocate(encoded_len);
16    char* p = EncodeVarint32(buf, internal_key_size);
17    memcpy(p, key.data(), key_size);
18    p += key_size;
19    EncodeFixed64(p, (s << 8) | type);
20    p += 8;
21    p = EncodeVarint32(p, val_size);
22    memcpy(p, value.data(), val_size);
23    assert(static_cast<size_t>((p + val_size) - buf) == encoded_len);
24    table_.Insert(buf);
25  }
```

As can be seen, the full length of the representation in bytes is computed (lines 9–14), then space allocated in the memory area (line 15), before the data is copied into the allocated buffer (lines 17–22). Finally, the skip-list is asked to store the pointer to the buffer (line 24). This adds the new entry to the correct place in the linked list, as defined by internal-key ordering. (The skip-list is set up with a custom comparator able to parse the byte-representation created by `Add(...)` and order entries based on the keys.)

When `Get(...)` is called, to search the memtable for a given key, the skip-list searches through its entries and compares the stored keys to the searched-for key. This is done through an `Iterator` interface provided by the skip-list. Once an entry is found, `Get(...)` verifies the found entry is valid by extracting and comparing the key from it. If it is, the type is extracted from the internal-key to determine if the entry is a valid key-value entry or a tombstone. This all looks like the following:

```
 1  bool MemTable::Get(const LookupKey& key, std::string* value, Status* s) {
 2    Slice memkey = key.memtable_key();
 3    Table::Iterator iter(&table_);
 4    iter.Seek(memkey.data());
 5    if (iter.Valid()) {
 6      // entry format is:
 7      //    klength  varint32
 8      //    userkey  char[klength]
 9      //    tag      uint64
10      //    vlength  varint32
11      //    value    char[vlength]
12      // Check that it belongs to same user key.  We do not check the
13      // sequence number since the Seek() call above should have skipped
14      // all entries with overly large sequence numbers.
15      const char* entry = iter.key();
16      uint32_t key_length;
17      const char* key_ptr = GetVarint32Ptr(entry, entry+5, &key_length);
18      if (comparator_.comparator.user_comparator()->Compare(
19              Slice(key_ptr, key_length - 8),
20              key.user_key()) == 0) {
21        // Correct user key
22        const uint64_t tag = DecodeFixed64(key_ptr + key_length - 8);
23        switch (static_cast<ValueType>(tag & 0xff)) {
24          case kTypeValue: {
25            Slice v = GetLengthPrefixedSlice(key_ptr + key_length);
26            value->assign(v.data(), v.size());
27            return true;
28          }
29          case kTypeDeletion:
30            *s = Status::NotFound(Slice());
31            return true;
32        }
33      }
34    }
35    return false;
36  }
```

Line 17 reads the `Varint32` to retrieve the internal-key length, and sets `key_ptr` to point to the data following the `Varint32`—in other words, the actual internal-key. Lines 18–20 compares the *user-key* portion of this internal-key to the searched-for key by excluding the last 8 bytes of the internal-key (refer to section 2.4.2 on internal-key format). If these match, the type-specifier encoded in the last byte of the internal-key sequence number is recovered (lines 22 and 23), and used to determine the type of value (lines 23–32). If it is a valid key-value pair (not a tombstone), a pointer to, and the size of, the value-part of the memtable-stored entry is calculated (line 25), and used to copy this data to the string passed to `Get(...)` for this purpose (line 26).

In summary, neither `Add(...)` nor `Get(...)` are particularly complicated methods: they primarily deal with serializing key-value data to a byte-array, and then defers efficient ordering of key-value pairs to the the skip-list. While the implementation is simple, the memtable as a whole makes up a very central part of LevelDB by being the primary in-memory storage structure.

### Skip-List

A skip-list is a probabilistic data structure, storing an ordered sequence of elements. It is designed to allow search-, insert-, and delete-operations in O(log N) average time.(Pugh, 1990)

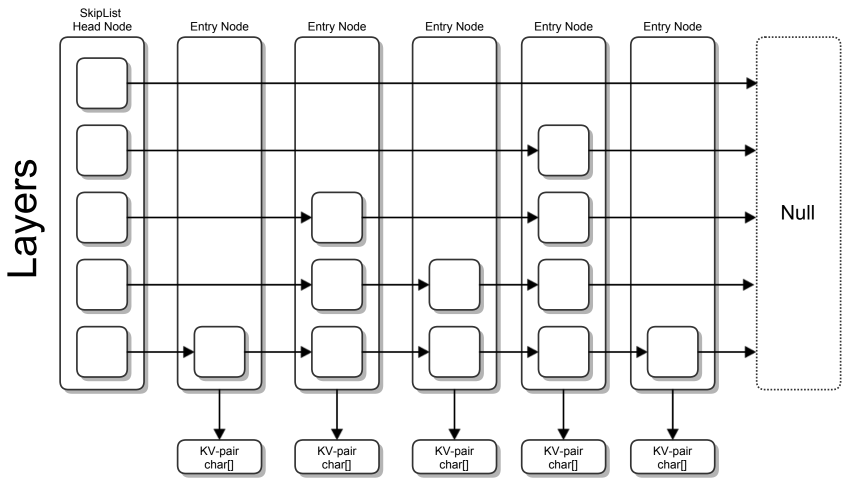

**Figure 2.7:** Memtable Skip-list

In terms of structure, it can be considered a type of ordered linked list—but where a hierarchy of subsequence links—each *skipping* over fewer elements—are stored instead of just one. This allows for uninteresting elements to be jumped over quickly when looking for a node. Figure 2.7 illustrates a skip-list as used by the LevelDB memtable.

During lookup, by considering the hierarchy of links from highest to lowest and knowing that the nodes in the skip-list are ordered, any pointer to a node less than the searched-for node can be immediately followed. For example, if the last node in figure 2.7 is the target node, the first non-null link in the start node (head) is in the second highest layer. The node it points to will then be found to be *less* than the target node, so the link can be immediately followed—skipping over nodes 1–3. Starting the process over from the linked-to node, the link in the bottom layers is eventually followed resulting in the target node being found.

When creating the skip-list structure, the number of links above the bottom layer (which is an ordinary linked list of all elements), is chosen probabilistically—the manner of which determines the skip-list's actual average-case time complexity. Once the number has been determined, the nodes preceding the added node has their links updated to point to this new node.

Average O(log N) lookups and insertions, along with straightforward ordered iteration, makes the skip-list structure particularly appropriate for the memtable. It is additionally both easy and efficient to support concurrent operations on the skip-list, something LevelDB takes advantage of to allow readers to operate on it without having to acquire any locks. Section 2.6 contains some additional reflections regarding this.

### 2.4.4 The Version and VersionSet Classes

To manage the set of on-disk sstables, LevelDB uses a `Version` class which maintains information about which sstables exists at each level along with compaction-supporting metadata. To illustrate this, the following is the relevant portion of its private data members (members used for reference counting, parent `VersionSet`, and membership in an intrusive linked list omitted):

```
// List of files per level
std::vector<FileMetaData*> files_[config::kNumLevels];

// Next file to compact based on seek stats.
FileMetaData* file_to_compact_;
int file_to_compact_level_;

// Level that should be compacted next and its compaction score.
// Score < 1 means compaction is not strictly needed.  These fields
// are initialized by Finalize().
double compaction_score_;
int compaction_level_;
```

A `Version` also has methods used to determine compaction-relevant information, such as which files in a level a given key-range overlaps. This is can be used to determine which files must be part of a compaction such that the non-overlapping invariant can be maintained. A `Version` is also responsible for performing lookups (`Get` operations) and creating iterators on the sstable-files it is responsible for:

```
// Append to *iters a sequence of iterators that will
// yield the contents of this Version when merged together.
// REQUIRES: This version has been saved (see VersionSet::SaveTo)
void AddIterators(const ReadOptions&, std::vector<Iterator*>* iters);

// Lookup the value for key.  If found, store it in *val and
// return OK.  Else return a non-OK status.  Fills *stats.
// REQUIRES: lock is not held
Status Get(const ReadOptions&, const LookupKey& key, std::string* val,
           GetStats* stats);
```

Each `Version` is owned and managed by a `VersionSet`, which is a top-level component of LevelDB. This is necessary since each `Version` is meant to be immutable once created; if changes to underlying data-files are performed a new `Version` is created and then becomes the new "current" `Version` as part of the `VersionSet`. This is part of the implementation details on how LevelDB internally handles the process described in section 2.3.7 on durability and recovery. The `VersionSet` also maintains information on which files-numbers to use for new files (for example log-files), so that the numbering is monotonically increasing, and what is the current maximum sequence number. Also among its responsibilities is tracking which `Versions` are still "live" even after having been superseeded—for instance because an iterator was created on it. Using this information it it can produce a report of all still “active” files—files referenced by any `Version` still live. This is used to find and delete orphaned sstables.

### 2.4.5 Sstable

Having already described the purpose of sstables, and some of the type of operations they allow for, this section will look in more detail on how a single sstable is structured.

The sstable file format depicted in figure 2.8. Each sstable contains a number of blocks, most of which are used to store the actual key-value entry data, along with block-type and a CRC32-number used to detect block-corruption. By partitioning the data into blocks, each block becomes an unit of fault-tolerance; data-corruption inside a block will only affect that block leaving others in the clear. The block-size is a configurable parameter,[10] but this is not a strict limit—only the cutoff where, once above it, LevelDB puts subsequent data in a new block. Each block can also optionally be compressed, further reducing its size from the block-size parameter. Whether or not a block is compressed is indicated by the `type`-byte following each block. (A compressed block is decompressed when loaded in to memory by LevelDB, so that its contents can be read.)

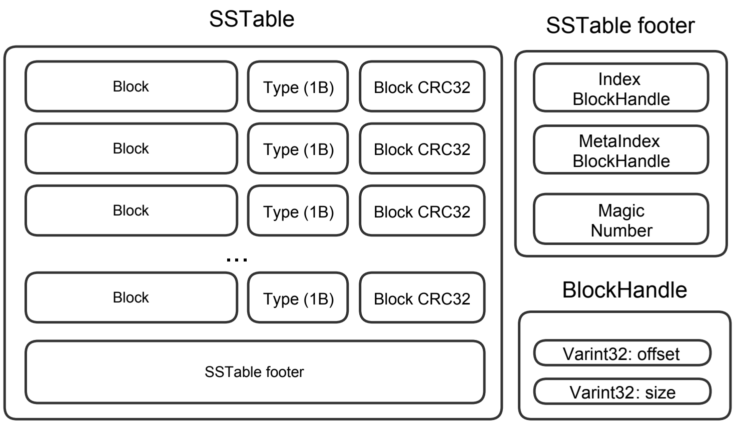

**Figure 2.8:** Sstable file format. The size and offset of each non-index-block is described in the index-blocks referenced from the footer.

The sstable footer is always the starting-point for lookups into an sstable: it contains both a magic-number indicating the file is indeed an sstable, and block-handles pointing to the block’s index- and meta-index-blocks. These are ordinary blocks in terms of structure, but instead of containing database key-value entries they contain one key-value pair for each block in the sstable. A block’s entry in the index has a key which is greater than or equal to the last key in the block while still smaller than the first key in the following block. The index-block can therefore be used to locate in which block a given database key-value

[10]LevelDB::Options::block_size

must be located if it exists in the sstable. The meta-index block works similarly, but is used to located meta-blocks containing information such as bloom-filters.

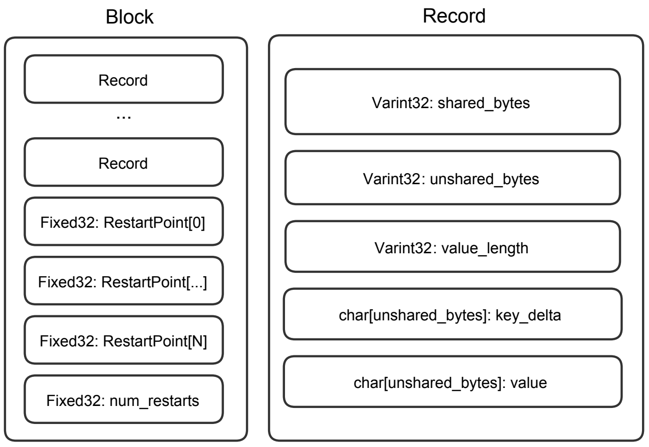

**Figure 2.9:** Format of an sstable-block and its record entries.

Figure 2.9 shows the format of each block. Each record in the block is a length-prefixed key-value pair, with additional fields to support prefix-compression of keys within the block. `shared_bytes` denote how many bytes the current key shared with the previous key, while `unshared_bytes` is the number of remaining non-shared bytes which is actually stored as part of the key. When iterating over records, LevelDB will use this information to decode the full key. The value size is stored in the `value_length` field. Should the value be of interest it can then be read, or else easily skipped over to located the next record.

To avoid having to always process the full block to locate a key (to resolve the prefix-compression), a number of *restart-points* are stored. These are records storing a complete non-prefix-compressed key. A restart point is created after a configurable number of keys (16 by default),[11] and there are direct pointer to them at the end of the block. A lookup can take advantage of this by binary-searching the restarts-points-list to locate the position of a given key. (This is possible since all records in both the sstable as a whole, and each block, are stored in sorted order.)

Given this fixed structure of both sstables and blocks, LevelDB includes iterator-classes which know how to navigate them to locate the record key-value pairs they store. These include an sstable-block-iterator which navigate the sstable's index-block to locate offsets and sizes for each block, and block-record-iterators which can navigate a block to locate its records. Combining these to form a `TwoLevelIterator`—named so because if goes through two levels of indirection to locate records: first the index-block to locate blocks, then the iterator returned from the block (a `leveldb:Block::Iter`) to locate

[11] LevelDB::Options::block_restart_interval

records—this iterator can then be used to directly get the key-value entries located in block-records without any knowledge of how the underlying storage is actually structured.

### 2.4.6 Concurrency

While LevelDB does not support multi-process concurrency—this is enforced through a lock-file which can only be held by one process at a time—the database object itself can safely be shared between multiple concurrent threads without any external synchronization.(LevelDB, 2012)

The way this is implemented is by requiring the member-variable mutex `DBImpl::mutex_` to be held when mutating most database-state. This is a rather coarse-grained level of synchronization, but the implementation attempts to hold this mutex for as short time as possible, and to release it during IO operations. If a long-running operation is started, additional synchronization might be set up so that `DBImpl::mutex_` can be released, but this is rarely needed. An example is described in section 2.4.9, on serialization of write-operations, where `DBImpl::mutex_` is released but new writer-operations are still blocked until the current write finishes. Some components also perform its own internal synchronization, for example the caches. This can be considered an example of finer-grained synchronization, and takes effect on read operations. As an example, after some initial coarse-grained synchronization when a read is started (see section 2.4.7), `DBImpl::mutex_` is released which allows operations from other threads to start. Since multiple read-operations can be in progress at once, the caches (which are used by all read-operations) need to perform synchronization outside of that provided by the `DBImpl::mutex_`. How this is done specifically is briefly covered in section 2.4.12 on cache-implementation, but boils down to simply locking the cache while modifications (such as updating a least-recently-used list) are done. Afterwards, while the data is actually processed, no locks are hold.

### 2.4.7 Reads

The implementation of read-operations involve performing lookups in three places: the current memtable, the immutable memtable, and the current `Version` (which maintains information about each level's sstables). It is therefore necessary these structures all stay alive for the duration of the lookup. LevelDB ensures this by using reference counting. The following is an excerpt from the start of the `DBImpl::Get` method:

```
MutexLock l(&mutex_);
SequenceNumber snapshot;
if (options.snapshot != NULL) {
  snapshot = reinterpret_cast<const SnapshotImpl*>(options.snapshot)->number_;
} else {
  snapshot = versions_->LastSequence();
}

MemTable* mem = mem_;
MemTable* imm = imm_;
```

```
11    Version* current = versions_->current();
12    mem->Ref();
13    if (imm != NULL) imm->Ref();
14    current->Ref();
```

On line 9–14, local pointers to these three structures are stored, and their reference count incremented. This creates the guarantee they will all be available until the reference count is decremented at the end of the function (when the lookup is complete). By storing local pointers, LevelDB is free to replace the original pointers while the read progresses—for instance by having the current memtable become the immutable memtable, or by having the current `Version` replaced by a new `Version` after compaction. The read-operation is allowed to be completely oblivious of this because it is only interested in the state as it was when the read started. The mutex locked on line 1 guarantees that the structures are in a consistent state when the function takes the references; if a memtable is currently being replaced, the mutex will block until everything is again consistent.

Since all reads are on a snapshot of the database, lines 2–7 either fetch the snapshot sequence number from the passed `ReadOptions`, or use the version number most recently used for any operation, as stored by the database's `VersionSet`.

This completes the read-preparation, and it subsequently progresses as follows:

```
 1    // Unlock while reading from files and memtables
 2    {
 3      mutex_.Unlock();
 4      // First look in the memtable, then in the immutable memtable (if any).
 5      LookupKey lkey(key, snapshot);
 6      if (mem->Get(lkey, value, &s)) {
 7        // Done
 8      } else if (imm != NULL && imm->Get(lkey, value, &s)) {
 9        // Done
10      } else {
11        s = current->Get(options, lkey, value, &stats);
12        have_stat_update = true;
13      }
14      mutex_.Lock();
15    }
```

First, the database-wide state-protecting mutex is released (line 3). Then each of the memtables (line 6 and 8), and finally the `Version` (line 11), are in turn given a chance to return a result. Should one of them do so is guaranteed to be newer than what any of the others could return, so the operation is done.

When `Version::Get` is called on line 11 to search the levels structure for data, it progresses as follows:

1. For each level:
    (a) Find all sstables in level which has a key-range containing the lookup-key. (Max 1 for all levels greater than level-0.)
    (b) For each candidate sstable, fetch it and look for an entry. (See section 2.4.5 on sstable structure.)

   (c) If entry was found, or found to have been deleted, return this. If not, try next level.

2. All levels have been looked at: return entry not found.

Once the lookup finishes, all structures which had their reference count incremented at the beginning of the read has their counts decremented. Depending on what has happened in other parts of the system while the read was in-progress, this can cause the final reference to any of them to be dropped, and the item finally deleted. The critical point to note here is how the read-operation, through reference counting, is able to force structures to stay alive while needed without this causing any inconsistencies in other parts of the database.

### 2.4.8 Writes

When a write is requested by the client, LevelDB first performs synchronization between it and other concurrent requests, so that they are serialized. This is the subject of section 2.4.9. Once a write is ready to be performed it is prepared, optionally logged, and finally written to the memtable. Below is the relevant source-code snippet from `DBImpl::Write` which does this:

```
 1  // May temporarily unlock and wait.
 2  Status status = MakeRoomForWrite(my_batch == NULL);
 3  uint64_t last_sequence = versions_->LastSequence();
 4  Writer* last_writer = &w;
 5  if (status.ok() && my_batch != NULL) {  // NULL batch is for compactions
 6    WriteBatch* updates = BuildBatchGroup(&last_writer);
 7    WriteBatchInternal::SetSequence(updates, last_sequence + 1);
 8    last_sequence += WriteBatchInternal::Count(updates);
 9    {
10      mutex_.Unlock(); // Safe to unlock during IO since writers are serialized
11      status = log_->AddRecord(WriteBatchInternal::Contents(updates));
12      bool sync_error = false;
13      if (status.ok() && options.sync) {
14        status = logfile_->Sync();
15        if (!status.ok()) {
16          sync_error = true;
17        }
18      }
19      if (status.ok()) {
20        status = WriteBatchInternal::InsertInto(updates, mem_);
21      }
22      mutex_.Lock();
23      if (sync_error) {
24        RecordBackgroundError(status);
25      }
26    }
27    if (updates == tmp_batch_) tmp_batch_->Clear();
28
29    versions_->SetLastSequence(last_sequence);
30  }
```

Before a write is handed to the memtable, LevelDB first checks whether or not the memtable has enough free space to accept the write (line 2). This is done through a call to `leveldb::DBImpl::MakeRoomForWrite`. It works in the following way:

1. If the number of level-0 files is getting large (in other words if the write-volume is so large that the compaction-process is not keeping up), wait 1 millisecond and check again. This delaying of writes is done to give the compaction process wall-clock time, as well as possibly CPU time, to catch up.
2. If the memtable reports a size below its configured maximum threshold, it has free space, so the function returns.
3. By getting to this step, the memtable must not have free space. It is then checked if an immutable memtable exists. If it does, there is nothing to do except wait until it has been compacted. A wait on a conditional variable triggered on compaction end is set up. Once it signals, LevelDB starts again from step 1.
4. If the number of level-0 files is above a configurable maximum, LevelDB waits until it has gone down. The wait in item 1 is an attempt to mitigate this eventual complete stop by delaying each write, but by getting here this has been shown to be insufficient—so LevelDB stops all writes temporarily until the compaction process has caught up. Once it does, it starts again from step 1.
5. By getting here, there is no immutable memtable, there is space in level-0, and the current memtable does not have enough space to accept the write. LevelDB thus makes the memtable an immutable memtable, creates a new memtable, and sets it up to service writes. Since an immutable memtable was created it is then also scheduled for compaction. The function returns now that the current memtable can receive the write.

At this point, the memtable is ready to receive the write. LevelDB looks up the current sequence number so that the new changes can have their own number calculated. It then builds a final `WriteBatch` through a call to `BuildBatchGround` (line 6). This can combine multiple write-requests into the current write, as described in section 2.4.9. Then the batch is updated with sequence number through a call to `WriteBatchInternal::SetSequence` (line 7). Finally, the number of changes is counted so that the next available sequence number, after the write is performed, can be calculated (line 8). On lines 10 to 17, the change is logged by simply serializing the textual representation of the `WriteBatch` (this includes all information including key- and value-data). The log is optionally synced to disk for durability (lines 13–18). As long as no errors are encountered, `WriteBatchInternal::InsertInto` is called to insert all changes in the memtable (line 20). Lastly, the database's next available sequence-number is updated through a call to `VersionSet::SetLastSequence` (line 29).

Once this process finishes, awaiting writers are signaled so that the next scheduled writer (if there is one) might proceed, and the process starts over again.

### 2.4.9 Serialization and Batching of Writes

LevelDB only allows a single writer to proceed at a time. This is implemented by keeping a list of to-be-performed writes, and having the requesting threads sleep until either the write is picked up by a thread preceding it in the list, or the requesting thread reached the front of the list. The first sections of `DBImp::Write` implements this logic:

```
 1  Status DBImpl::Write(const WriteOptions& options, WriteBatch* my_batch)
 2  {
 3    Writer w(&mutex_);
 4    w.batch = my_batch;
 5    w.sync = options.sync;
 6    w.done = false;
 7
 8    MutexLock l(&mutex_);
 9    writers_.push_back(&w);
10    while (!w.done && &w != writers_.front()) {
11      w.cv.Wait();
12    }
13    if (w.done) {
14      return w.status;
15    }
16    [...]
```

First, a `Writer` object is created to represent the to-be-performed operation (lines 3–6) and `DBImpl::mutex_`, the mutex protecting most of the database's internal state—including the `DBImpl::writers_` list—is locked (line 8). This allows the current thread to append itself to the list (line 9). What happens next depends on whether or not another write is already in progress: if one is, the current operations is *not* the first element in the list (the "current" or "active" writer), so it releases the mutex and waits (lines 10–11). Otherwise, it is free to proceed. Waiting writers are notified through a conditional variable on changes to the list, at which points it again checks if it should become the active writer (though only after first acquiring `DBImpl::mutex_`).

Once a writer is ready to proceed, it continues as described in section 2.4.8.

#### Write Batching

One optimization LevelDB does with this list of writers is to, when certain conditions are met, combine waiting write-operations into the current write. This is the purpose of checks regarding `w.done` when waking up from waiting: a waiting writer's changes can have been picked up by a writer preceding it in the list. By batching writes LevelDB is able to decrease the resulting overhead from having to process a larger number of separate write-operations.

The logic for writer-combining is performed by the line
`WriteBatch* updates = BuildBatchGroup(&last_writer);` executed shortly after a writer becomes the current active writer. The function will append changes requested by other writers to the current write as long as 1) the other write operation is set to be synchronous[12] while the active one is not, and 2) the combined operations are below a maximum size (measured in the internal textual representation of the `WriteBatch`).[13]

[12]Meaning it performs logging of the changes to the memtable.

[13]Maximum size is set to around 8 MB, unless the current operations is below 1 MB, in which case the maximum is set to around current_size + 1MB. This is to not slow down small writes too much by combining it

No matter if the current write-operations ended up picking up more changes or not, it subsequently proceeds as a normal write as covered by section 2.4.8.

### 2.4.10 Atomic Updates

Users of LevelDB are able to atomically apply a set of updates by creating and passing a `WriteBatch` object to `DB::Write`. An example of this is the following:

```
    leveldb::WriteBatch batch;
    batch.Delete(key1);
    batch.Put(key2, value);
    leveldb::Status s = db->Write(leveldb::WriteOptions(), &batch);
```

The criteria for atomicity in this context is that, to concurrent LevelDB clients, all changes in the batch will appear to be applied at the same time or all not be applied at all. How this is implemented is explained next.

As shown in section 2.4.8, a `WriteBatch` is already used internally by LevelDB for all write-operations (for example `Put` and `Delete`). To illustrate, `DB::Put` is implemented as follows:

```
Status DB::Put(const WriteOptions& opt, const Slice& key, const Slice& value) {
  WriteBatch batch;
  batch.Put(key, value);
  return Write(opt, &batch);
}
```

. . . in other words, by creating a `WriteBatch` consisting of a single operation. This batch is then processed internally and fails or succeeds as a unit.

By having `Get` operations implicitly work on a snapshot of the current state—as defined by the maximum sequence number when it started—and `Write` operations (i.e. the processing of a single `WriteBatch`) not incrementing this sequence number before the write is complete, the end result is all operations on a single `WriteBatch` all become visible as an unit the moment the write finishes. Applying a set of updates atomically are therefore not a "special case" for LevelDB—it is rather the default mode of operations, and single-item updates simply use this framework. This can be contrasted with other systems where single updates might be considered the default, and transactions are used to make a number of these appear atomic. It is worth noting how this is very much an implementation detail; the LevelDB API differentiates between the two cases (by providing separate API calls for `Put` and `Delete`) whereas they are all handled identically as "writes" internally.

with large writes.

### 2.4.11 Compactions

When LevelDB detects a compaction *might* be necessary (or beneficial), for example after creating an immutable memtable or due to the actions of a completing compaction, it calls `DBImpl::MaybeScheduleCompaction()`. This method check whether or not a compaction is necessary, and if so schedules it to be done in a separate thread:

```
void DBImpl::MaybeScheduleCompaction() {
  mutex_.AssertHeld();
  if (bg_compaction_scheduled_) {
    // Already scheduled
  } else if (shutting_down_.Acquire_Load()) {
    // DB is being deleted; no more background compactions
  } else if (!bg_error_.ok()) {
    // Already got an error; no more changes
  } else if (imm_ == NULL &&
             manual_compaction_ == NULL &&
             !versions_->NeedsCompaction()) {
    // No work to be done
  } else {
    bg_compaction_scheduled_ = true;
    env_->Schedule(&DBImpl::BGWork, this);
  }
}
```

Once the thread starts, it eventually calls `DBImpl::BackgroundCompaction()`. From here, either `CompactMemTable()` is called—if there is an immutable memtable awaiting compaction—or sstable compaction is prepared. Both cases are covered below.

#### Memtable Compaction

Memtable compactions are always prioritised to avoid the current memtable filling up while the immutable memtable is still awaiting compaction (which will stall LevelDB). The process itself is a relatively straightforward:

1. Write the immutable memtable's contents to an sstable.
2. Create a new `Version` object which includes this sstable in level-0 and does not include the immutable memtable's log.
3. Install this `Version` as the current `Version` object.
4. `Unref()` the immutable memtable (potentially deleting it), and set `DBImpl::imm_` to `NULL` indicating it has been compacted.
5. Delete the immutable memtable's now obsolete log file.

Step 1 is handled by `WriteLevel0Table()`. It involves extracting key-value pairs from the immutable memtable (by iterating over it) and then creating an sstable from these entries.

Steps 2 and 3 are relatively straightforward, but includes some complexities with making sure the new `Version` is durably logged to disk so that no data loss is possible. (If the new `Version` is not durably written, but the memtable's log file deleted, LevelDB could end up crashing, recovering the old `Version` on startup, but then not finding the memtable's log.) One additional optimization LevelDB tries here is to place the sstable in a level lower than 0 if this can be done without invalidating any of the levels-structure's invariants.

In step 4, changing `DBImpl::imm_` directly is safe as LevelDB makes sure `DBImpl::mutex_` is held before doing so. (It is actually held during the entire process, except for the relatively expensive creation of the sstable in step 1.) Step 5 frees the disk-space taken up by the log; it is no longer needed as the memtable's contents are now durably on disk.

### Sstable Compaction

Sstable compaction is the more complicated compaction-type, as it has to handle removal of superseded key-value entries and entry deletions, while still keeping the necessary old versions for snapshot support.

To identify which sstables are to be merged, `DBImpl::BackgroundCompaction()` calls `VersionSet::PickCompaction()`. This method determines which sstables to merge in the following way:

1. If a level has gone above its threshold size, this will have been recorded by the `Version` class when the sstables which made it go above the threshold were added to it. If there is such a level, an initial sstable is chosen by looking at the key stored in `VersionSet::compact_pointer_[level]`, or the first file if this is not set or no matching sstable is found.
2. Otherwise, if an sstable has been pointed out as a compaction candidate—for example by being the first considered sstable for many entry lookups without returning a result— `Version::file_to_compact_` is non-`NULL`, and this sstable is select as the initial sstable.
3. If the to-be-compacted sstable is in level-0, other level-0 sstables which overlap its key-range are also added to the list of to-be-compacted sstables.
4. The overlapping sstables in the compacted-to level is identified. Based on the key-range of these files, additional files in the *from-level* is added if this do not increase the number of files to merge with in the *compacted-to* level.
5. `VersionSet::compact_pointer_[level]` is updated so a different range is considered the next time the level is to be compacted.

The eventually result is a list of files at level-X and level-X+1 to merge, along with some additional information, in a `Compaction` object.

Once all input-files are gathered, the actual compaction is done in `DBImpl::DoCompactionWork`. Here the lowest snapshot sequence number is found, and a merging iterator—pulling key-value pairs from all input-files—is created. Each pair, in order determined by the internal-key representation (section 2.4.2), are looked at, and the decision whether to keep or drop the pair is done as follows:

1. The first seen entry with a given user-key is kept. (This is the most recent entry with that key.) An exception is deletion-markers, which can be deleted if there is guaranteed to not be any entries with the same key in higher-numbered levels.

2. The next entry with an already seen user-key is deleted, unless it needs to be kept for snapshot-support reasons (see section 2.4.13 for details).

A non-dropped entry is added to the sstable currently being constructed:

```
if (!drop) {
  // Open output file if necessary
  if (compact->builder == NULL) {
    status = OpenCompactionOutputFile(compact);
    if (!status.ok()) {
      break;
    }
  }
  if (compact->builder->NumEntries() == 0) {
    compact->current_output()->smallest.DecodeFrom(key);
  }
  compact->current_output()->largest.DecodeFrom(key);
  compact->builder->Add(key, input->value());

  // Close output file if it is big enough
  if (compact->builder->FileSize() >=
      compact->compaction->MaxOutputFileSize()) {
    status = FinishCompactionOutputFile(compact, input);
    if (!status.ok()) {
      break;
    }
  }
}
```

A `leveldb::TableBuilder` is used to construct the sstables themselves, and once the table reaches a threshold size a new sstable is started.

Eventually this process finishes, and the changes made by the compaction is registred by producing a new `Version` object. To do this, the sstables used as *inputs* to the compaction are removed from the levels where they were located, and the *outputs* (the newly produced sequence of sstables) are added to the compacting-to-level. This new `Version` is then logged and installed in the usual way.

### 2.4.12 Caching

LevelDB has two caches internally; an (ss)table-cache and a block-cache. Both have a configurable size; the table-cache can contain a number of open files (default 1000), while the block-cache has a given maximum size (8 MB by default).

### Table Cache

The table-cache is structured as a layer abstracting away all direct access to sstables. This is so it can retain full control over their access. The cache provides three public methods:

```
Iterator* NewIterator(const ReadOptions& options,
                      uint64_t file_number,
                      uint64_t file_size,
                      Table** tableptr = NULL);

Status Get(const ReadOptions& options,
           uint64_t file_number,
           uint64_t file_size,
           const Slice& k,
           void* arg,
           void (*handle_result)(void*, const Slice&, const Slice&));

void Evict(uint64_t file_number);
```

In other words, the creation of an iterator for a table, a get-operation on a table, and the eviction of a table from the cache. The first two are rather self-explanatory. The latter is used when a table is orphaned, for example after having its content compacted to a new file, and scheduled for deletion.

The methods identifies tables by using a numbered identifier, stored as an `uint64_t`. The information about what each file contains, and its identifier, is stored by the `Version` objects. When a request for information in an sstable is to be done, the table-cache `Get` method is called with the file-number as part of its arguments. The requested sstable is then loaded if it is not already in the cache, and the table-cache handles lookups in it. The final two parameters to `Get` are a callback function pointer and an associated callback user-pointer used to potentially return a found key-value pair to the caller.

Both the table-cache and block cache use the same underlying cache-implementation, but the difference is in what that cache stores. In both cases, a `void*` (void-pointer) is returned, and for the table-cache this points to a `TableAndFile`-structure which itself contains pointers to both a `Table`-object and a `RandomAccessFile`-object (which can be used to read the sstable-file directly).

### Block Cache

The block-cache, as opposed to the table-cache, is entirely client-configurable. Part of the `leveldb::Options`-argument passed to `leveldb::Open` when creating the database, it is used to cache and retrieve single sstable blocks. By default, if the user does not pass a custom cache, a 8MB least-recently-used cache is created.[14]

The block-cache is internally only used by `Table` objects when a block is requested from it (either directly or through an iterator). The following source code extract from `Table::BlockReader` shows how it is used:

[14] `SanitizeOptions` in db_impl.cc

```
 1  BlockContents contents;
 2  if (block_cache != NULL) {
 3    char cache_key_buffer[16];
 4    EncodeFixed64(cache_key_buffer, table->rep_->cache_id);
 5    EncodeFixed64(cache_key_buffer+8, handle.offset());
 6    Slice key(cache_key_buffer, sizeof(cache_key_buffer));
 7    cache_handle = block_cache->Lookup(key);
 8    if (cache_handle != NULL) {
 9      block = reinterpret_cast<Block*>(block_cache->Value(cache_handle));
10    } else {
11      s = ReadBlock(table->rep_->file, options, handle, &contents);
12      if (s.ok()) {
13        block = new Block(contents);
14        if (contents.cachable && options.fill_cache) {
15          cache_handle = block_cache->Insert(
16              key, block, block->size(), &DeleteCachedBlock);
17        }
18      }
19    }
20  }
```

First, on lines 3-6, an unique 16-byte representation of the block is constructed. This includes an id for the table it is part of, as well as the offset the block has in the sstable. This is then used as lookup-key in the cache (line 7). A successful lookup will return a *handle* from the cache, which can subsequently be used to also get the actual value. On line 9, a pointer to the block in question is found in this way. Should the cache not contain the block, a `NULL` handle is returned. In this case the block is manually read in from disk and a `Block` object is constructed to represent it (lines 11–13). Unless the block is marked as non-cacheable,[15] or if the read is marked as a non-cache-filling read,[16] the block is inserted into the cache (line 15).

### LRU-cache implementation

LevelDB uses an implementation of a sharded[17] least-recently-used (LRU) cache for the table- and block-cache, by default. This section will describe how this cache is implemented.

As mentioned in section 2.4.6 on concurrency in LevelDB, the caches provide their own synchronization. Because multiple operations can potentially be in-progress at the same time, this makes the caches potential hotspots in the case of, for example, many simultaneous read-operations. For likely this reason, LevelDB opts to not use a single LRU cache, but rather partition the key-space into a number of separate LRU caches. By default, each such "sharded LRU cache" consists of 16 individual LRU-caches, where the actual LRU cache to use is selected by hashing the lookup-key and looking at the 4 most significant bits. Because the LRU caches themselves simply lock a mutex before each operation, this helps avoid the complete serialization of cache-operations one would experience should a single LRU-cache be used for all operations instead of this sharded scheme.

[15]This can happen if the block-data is returned as a direct pointer to some data (e.g, a memory-mapped area), rather than memory allocated by LevelDB specifically to hold the block-contents.

[16]This is a client-specified option, which can for example be set on an iterator-driven bulk read.

[17]Note: *sharded*, not *shared*.

The LRU cache itself is implemented by maintaining an internal hash-table for fast lookups by key, and a doubly-linked-list of all items to maintain the least-recently-used order. On a successful lookup the item it simply removed from its place in the list, and inserted in front:

```
Cache::Handle* LRUCache::Lookup(const Slice& key, uint32_t hash) {
  MutexLock l(&mutex_);
  LRUHandle* e = table_.Lookup(key, hash);
  if (e != NULL) {
    e->refs++;
    LRU_Remove(e);
    LRU_Append(e);
  }
  return reinterpret_cast<Cache::Handle*>(e);
}
```

Each item inserted “charges” the cache by a given number: for the table-cache the charge is 1, while for the block-cache it is the byte-size of the block. After an insert which takes the cache’s charged usage above its capacity, the LRU-list is iterated over in least-recently-used order and items removed until the charged usage drops to below the capacity—or the list wraps:

```
  while (usage_ > capacity_ && lru_.next != &lru_) {
    LRUHandle* old = lru_.next;
    LRU_Remove(old);
    table_.Remove(old->key(), old->hash);
    Unref(old);
  }
```

What is worth nothing from the above code is that no changes to the `usage_` variable is done directly. This happens in the `Unref`-method, but only when the item is actually deleted—meaning when its reference count drops to zero. This is a consequence of the cache not being able to forcefully delete an item, as a positive reference-count indicates the item is still being used somewhere else in the system. Once the usage is finished, the caller having fetched the item from the cache, is expected to call the cache’s `Release`-method so that the reference count is decremented. When this happens on an item still in the LRU-list, the reference count stays positive (because it is referenced by the list) and the item stays in the cache and is not deleted. If the item should be removed from the cache (because it has been removed from the LRU-list), the reference count will eventually hit zero as it is released by all users—and the item is then deleted. Only then is the charged usage of the item decremented from the cache.

The “capacity” of the cache is therefore not an upper bound on usage, but rather the point where the cache stops trying to artificially keep items alive even though they are not currently used. So for example, if the system need more blocks in memory than the cache has capacity for, the cache can not stop it—it can only decline to keep these blocks alive after they are finished being used.

### 2.4.13 Snapshots and Iterators

The implementation of snapshots is, as the description in the architecture section 2.3.6 might cause one to suspect, not particularly complex. Since a snapshot is simply a sequence number indicating where it takes place in the database's timeline, the creation and deletion of snapshots only amount to the following:

```
const Snapshot* DBImpl::GetSnapshot() {
  MutexLock l(&mutex_);
  return snapshots_.New(versions_->LastSequence());
}

void DBImpl::ReleaseSnapshot(const Snapshot* s) {
  MutexLock l(&mutex_);
  snapshots_.Delete(reinterpret_cast<const SnapshotImpl*>(s));
}
```

...where the `New` method is used to add the current sequence number to the end of a doubly linked `snapshots_` list, and the `Delete` method removes the number from the list. This way—given that the current sequence number is monotonically increasing—the oldest snapshot is always at the beginning of the list, and for snapshot-support it is sufficient to only keep alive the *first* entry of a key with a sequence number less than this number, as well as newer entries, while all older entries can be deleted without issue. The logic for this when performing compaction looks like the following:[18]

```
      if (last_sequence_for_key <= compact->smallest_snapshot) {
        // Hidden by an newer entry for same user key
        drop = true;    // (A)
      } else if (ikey.type == kTypeDeletion &&
                 ikey.sequence <= compact->smallest_snapshot &&
                 compact->compaction->IsBaseLevelForKey(ikey.user_key)) {
        // For this user key:
        // (1) there is no data in higher levels
        // (2) data in lower levels will have larger sequence numbers
        // (3) data in layers that are being compacted here and have
        //     smaller sequence numbers will be dropped in the next
        //     few iterations of this loop (by rule (A) above).
        // Therefore this deletion marker is obsolete and can be dropped.
        drop = true;
      }
```

`last_sequence_for_key` is a valid number if an entry with the same key has already been processed,[19] and `compact->smallest_snapshot` is the smallest number from the `snapshots_`-list above. If there is no snapshots, `smallest_snapshot` will be set to the next free sequence number so that entries are artificially kept alive.

For iterators, the primary requirement is that memtables and sstables are kept alive for the duration of the iterator. This is done by using reference counting. From `DBImpl::NewInternalIterator`:

[18] `DoCompactionWork` in file `db_impl.cc`

[19] Otherwise it is a large number, so the initial test fails.

```
 1  // Collect together all needed child iterators
 2  std::vector<Iterator*> list;
 3  list.push_back(mem_->NewIterator());
 4  mem_->Ref();
 5  if (imm_ != NULL) {
 6    list.push_back(imm_->NewIterator());
 7    imm_->Ref();
 8  }
 9  versions_->current()->AddIterators(options, &list);
10  Iterator* internal_iter =
11      NewMergingIterator(&internal_comparator_, &list[0], list.size());
12  versions_->current()->Ref();
13
14  cleanup->mu = &mutex_;
15  cleanup->mem = mem_;
16  cleanup->imm = imm_;
17  cleanup->version = versions_->current();
18  internal_iter->RegisterCleanup(CleanupIteratorState, cleanup, NULL);
```

Iterators which the to-be-created global iterator will merge data from are collected from the current memtable, the immutable memtable and the current `Version` (line 3–9). All these are then `Ref()`'d so that they are kept alive for the lifetime of the iterator. The created iterator, `internal_iter`, is then set up with cleanup-information so that all necessary `Unref()`s are called once it is deleted (lines 14–18).

This has the effect that neither of the memtables, nor any sstable-files managed by the `Version` (by virtue of the `Version` object itself staying alive) will be deleted as long as the iterator is alive. This enables it to provide a consistent view of the database contents as it were when the iterator was created, even as the current state of the database changes.

### 2.4.14 Recovery

In section 2.3.7, it was described how LevelDB makes sure all changes are done in such a way as to be recoverable in the event of crash or system failure.

The two reasons for change were additions to the memtables and the addition of a new `Version` (after compaction). Both of these are handled by logging the changes to disk before applying them. This section will look at how recovery is done for both these cases.

After the database is opened, `DBImpl::Recover()` is called to restore it to a valid state. This is done in two steps: first recover the current `Version`, then the memtables.

#### Version

Recovery of the current `Version` is done by `VersionSet::Recover()`. It does the following:

1. Reads the file named *CURRENT*; this contains the name of the file used to log changes to the `Versions`.

2. Parses the log file, applying logged edits as they are found. The log will always start with all necessary information from the `Version` which was current when the log was created, while after this only subsequent changes are logged. This eventually recovers the `Version` as it was after the last successfully written log-entry.

The main workhorse during this operation is the the `leveldb::log::Reader` class, which decodes records from the log file. These records are arbitrary byte-arrays, but in this case they represents serialized versions of the changes made to the `Version` in the form of a `VersionEdit` object. The edits are eventually combined to the final `Version` by a `Builder` class made specifically for this purpose.

### Memtables

With the latest `Version` recovered, it is possible to discover which on-disk memtable log files were live at the time of a shutdown or crash. (As opposed to log files which are orphaned, but not yet deleted.) This is done by assigning logfiles monotonically increasing identifiers, and storing the identifiers of the last memtable-log successfully converted to an sstable. Any memtable log files with identifiers greater than this number therefore needs to have their contents recovered.

By considering all files in the database folder, identifying which ones are log files by their name, and then looking at the identification number, all to-be-recovered log files are located. Each are then passed to `DBImpl::RecoverLogFile()` in order of increasing identifier number. This method, similarly to when the `Version` is recovered, use the `leveldb::log::Reader` class to parse record out of the log file. Each record is a write-operation previously applied to the memtable the log file represents, and this operations is simply replayed to a newly created memtable. This eventually recovers the entire memtable as it was after the last successfully written log record.

As the memtables are recovered, they are subsequently written to level-0 sstables by `WriteLevel0Table()`, and the successful recovery of it (and the addition of the new sstable) is recorded in the current `Version`.

### Final Steps

After successfully recovering both the current `Version` and the memtables, this recovered database state is finally itself durably written to disk to avoid having to redo the same recovery operations later. This involves creating a new `Version` log file, and updating the *CURRENT* file to point to this file. Once this is done, the old log files are deleted by calling `DeleteObsoleteFiles()`.

This concludes the description of how LevelDB does recovery, and shows how it is able to successfully recover in-memory content (`Version` state and memtables), by logging changes to these to disk.

## 2.5 Extensions and Related Projects

LevelDB's open-source nature has allowed for it to be extended and improved by third parties unrelated to Google. Three of these are briefly covered below.

### HyperDex HyperLevelDB

HyperDex is a NoSQL key-value store. It lists a "rich API, strong consistency, and fault tolerance" as its key features.(HyperDex, 2014) For its data storage engine it uses HyperLevelDB, an open-source fork of LevelDB. HyperLevelDB aims to improve on LevelDB in two primary ways:(HyperDex, 2013)

**Parallelism:** HyperLevelDB removes the complete serialization of writer threads by having them first agree on the ordering, and then independently apply the writes in a manner consistent with this order. This helps improve both write-throughput and multi-threaded utilization.

**Compaction:** LevelDB's approach to compaction does not take into account the ratio of data in the receiving-level which must be rewritten because of the compaction to data actually moved across levels. HyperLevelDB tries to minimize this overhead by selecting what data to compact using this as a metric, instead of progressively moving through the level key-space as LevelDB does by default. By reducing the amount of data written, this enables the system to perform additional compactions—increasing total throughput.

### Baso LevelDB

Basho Technologies, developers of the Riak NoSQL database, maintains their own LevelDB fork and provides it as one of the available storage backends for Riak.(Basho, 2014a) The branch is, as of May 2014, in active development. It has integrated a number of changes deemed to improve performance. This includes both larger architectural changes, such as increasing the number of concurrent compaction threads and the number of levels allowed to contain overlapping sstable-ranges, as well as more low-level changes such as utilizing hardware features to calculate CRC checksums.[20][21][22]

### Facebook RocksDB

RocksDB is a project started at Facebook which forks LevelDB with the aim to create a database especially suitable for running on flash storage. RocksDB additionally extends

[20]https://github.com/basho/leveldb/wiki/mv-hot-threads
[21]https://github.com/basho/leveldb/wiki/mv-level-work3
[22]https://github.com/basho/leveldb/wiki/mv-verify-compactions

LevelDB with a number of features, including the ability to take full backups, and a replication system.(Jin, 2014)

## 2.6 Discussion

### Relation to Google Bigtable and LSM Trees

Many of the concepts used by LevelDB, including memtables and sstables, were introduced by Google in their Bigtable paper in 2006.(Chang et al., 2006) In Bigtable, the combination of a memtable and multiple sstables (all in a single level) is used to form a "tablet". This table is then subsequently used to store consecutive rows of a Bigtable table. Merging compactions are performed between sstables in the background, here based on their size (smaller sstables are merged together first). This is to limit the total number of sstable-files a read-operation has to look at to find a result, while still keeping each compaction relatively cheap.

As acknowledged in the Bigtable paper, this usage of memtables and sstables is analogous to how updates to index data is done in a Log-Structured Merge (LSM) Tree.(Chang et al., 2006)(O'Neil et al., 1996) In LSM-parlance, the memtable can be considered the in-memory $C_0$ component, while the set of sstables forms the on-disk $C_1$ component. The similarities quickly end however, and it is primarily the ideas of buffering data in memory in a sorted manner before writing it to disk sequentially, and how on-disk and in-memory data is merged on reads, which is borrowed from the LSM tree by Bigtable.

LevelDB extends the ideas from Bigtable by adding subsequent layers—or levels—of sstables below the initial level. Merging between these layers is done in in a manner similar to a multi-component (2+) LSM-tree. This has the effect of long-lived key-value pairs being migrated to lower levels over time, increasing the lookup-time for such a key as all levels preceding its location are checked before it is found. LevelDB therefore performs best if most lookups are for recently added, rather than old, data. By limiting the number of possible level-0 sstables, the maximum number of disk seeks needed to locate an arbitrary entry still has a well-defined upper bound however.

Both of these approaches to compaction, the Bigtable-inspired single-level size-tiered approach, and LevelDB's "leveled compaction"-approach, are available as selectable options in the Apache Cassandra database.[23] Comparing the two, they found that the size-tiered approach has multiple problems with regards to fragmentation of related data over multiple sstables (requiring a seek in each to acquire the complete set—this is analogous to how all level-0 sstable files must be considered on reads in LevelDB), and substantial amounts of disk space wastage. The leveled approach was found to solve these issues, but at the cost of roughly twice as much IO as the size-tiered approach. The preferred option was therefore found to depend on database usage, especially the ratio of inserts compared to reads.(Ellis, 2011)(Hobbs, 2012)

Similarly to an LSM-tree, both LevelDB and Bigtable incurs little IO-cost on inserts.[24] Anticipating more detailed benchmark-comparisons in chapter 4, both random and sequential

[23]Cassandra borrows many ideas from Bigtable, and similarly to LevelDB also use the concepts of memtables and sstables in its implementation.

[24]None if memtable-changes are not logged for durability.

writes are likely to perform well for this reason. Random reads is by far the slowest operation LevelDB can perform, as the majority of data in a large database is going to be located in higher-numbered levels—which are not looked at before lower-numbered levels are first checked. Sequential reads will receive an advantage from the sorted nature of sstables, meaning read X+1 is likely to have its target block located in the block-cache, saving IO costs.

### Choice of Skip-List to implement Memtable

While the Bigtable paper doesn't specify which structure is used to maintain the information in its memtable, the LSM-tree paper suggests a 2-3 tree or AVL-tree structure for the analogous $C_0$ component. LevelDB forgoes such a balanced tree entirely and instead opts to use a skip list. Invented by William Pugh, his 1990 article compares the skip list favorably to such balanced trees, and also references an algorithm allowing multiple processors to concurrently update a skip list in shared memory.(Pugh, 1990) This algorithm is additionally referenced to as "much simpler" than algorithms for concurrent balanced trees. More recent papers have also introduced efficient concurrent implementations, both lock-free and otherwise.(Fraser, 2004)(Maurice Herlihy, 2012) The LevelDB implementation only allow the memtable to be *updated* by one thread concurrently (using locks), but it does allow read-operations to progress concurrently in a lock-free manner. This ease of allowing this, along with the skip-list's general performance characteristics, is a likely reason for why a skip-list was chosen over some kind of balanced binary tree.

### Logging

When maintaining a write-ahead-log for memtable durability, LevelDB logs the full key-value data in its entirety. Parallels can be drawn between this and the idea of "physical logging".(Gray, 1993) While some kind of delta encoding when values for the same key is changed is conceivably possible—which could form the base for more "logical"-style logging—this is likely to be both very expensive (as old values need to be looked up during writes) and hard to integrate. Not being able to do this has notable performance implications, however, especially if large values are used, as all key-value pairs are always written twice—first to the log, then later to an sstable. Still, the benefit such physical-style logging is that recovery is greatly simplified.

### Stalls

One major issue LevelDB is seen to run into, and something forks of it often try to address with different, but related, approaches, is intermittent spikes in latency during write operations. This happens if both memtables (current and immutable memtable) fill up while the background thread is busy with a long-running compaction. In such a case, LevelDB simply stalls until the compaction process catches up. Most forks tries to handle this by increasing the number of compaction threads, and possibly also by dedicating a small set of threads exclusively to flushing memtables once they fill up.(Jin, 2014)[25] Since "stock" LevelDB only use a single compaction thread, write-heavy use cases is likely to see varying response-times and stalls (for writes) should the compaction process be unable to keep up.

[25]https://github.com/basho/leveldb/wiki/mv-hot-threads

Chapter 3

# Case Study: Symas Lightning Memory-Mapped Database (LMDB)

## 3.1 Introduction

Lightning Memory-Mapped Database (LMDB)[1] is an embedded key-value data store developed by Symas, written in C.(Symas, 2014) It was developed for the OpenLDAP-project[2] with the aim of replacing BerkeleyDB (BDB)[3] as its storage backend. For this reason, LMDB's API is inspired by BerkeleyDB, but otherwise the design and implementation is such as to perform much better in read-heavy situations.

### Background

OpenLDAP started using BDB as part of its storage backend in 2002.(Chu, 2012) To achieve sufficient performance they found it necessary to add multiple levels of caches on top of BDB. This ended up being a complex process: the caches themselves required careful tuning and the backend code became difficult to improve and maintain as locks had to be introduced to maintain coherency between the caches and BDB itself. Though the caches improved performance by an order of magnitude in general,(Chu, 2013b) they were sometimes not beneficial depending on configuration properties—in the worst cases they

[1]Previously MDB, but renamed to avoid confusion with other software.
[2]http://www.openldap.org/
[3]http://www.oracle.com/us/products/database/berkeley-db/overview/index.html

were completely inefficient, thus only wasting memory and processing effort by duplicating data.(Chu, 2013b) Summed up, the developers found the system "difficult to configure, difficult to optimize, and extremely labor intensive to maintain".(Chu, 2012)

To improve the situation, they seeked a solution where no complex cache management was necessary—they expected the filesystem cache to be sufficient—and used multi-version concurrency control (MVCC) to avoid the need to perform any locking when reading data. Surveying the database library landscape, they found no solution with the desired characteristics and thus decided to write their own.(Chu, 2013b)

## Features

Symas highlights the following as some of LMDB's major features:(Chu, 2013b)(Symas, 2014)

- Key/Value store implemented using B+trees: ordered-map interface with sorted keys; support for range lookups.
- Fully transactional: ACID semantics with MVCC. Readers never blocking writers, and writers never blocking readers. Write transactions fully serialized, meaning also deadlock-free.[4]
- Support for multi-thread and multi-process concurrency.
- Use of memory-mapped files allowing zero-copy lookup and iteration.
- Design supporting recovery-free restart.
- No background cleanup or compaction process required.

More advanced features include support for storage of multiple databases in a single file, nested transactions, and storage of multiple values for a single key.

With LMDB's genesis as a backend for OpenLDAP, it aims to provide a heavily read-optimized data storage solution. How these features are achieved and implemented, and what drawbacks and tradeoffs are made in pursuit of this, will be covered in subsequent sections.

[4]Whether or not being "deadlock-free" can be considered a feature, when it is achieved through complete serialization of write transactions, is a decision left up to the reader. In any case, having such single-writer-at-a-time semantics is common for embedded key-value store.

## 3.2 Usage Example

This section illustrates the basic usage of LMDB. Note that error handling is not shown; normally `rc` would be checked after each operation where it appears. Full documentation for the API can be found online.[5]

First a database environment is created:

```c
int rc; // return code
MDB_env *env;
rc = mdb_env_create(&env);
rc = mdb_env_open(env, "./testdb" /*path*/, 0, 0664);
```

Next a transaction is created, and a database opened as part of the transaction:

```c
MDB_txn *txn;
rc = mdb_txn_begin(env, NULL /*parent transaction*/, 0, &txn);
MDB_dbi dbi;
rc = mdb_open(txn, NULL /*database name*/, 0, &dbi);
```

To insert an item into the database, `mdb_put(...)` can be used as follows:

```c
int key_num = 123;
char data[] = {"Data to put"};
MDB_val key, data;
key.mv_size = sizeof(int);
key.mv_data = &key_num;
data.mv_size = sizeof(data);
data.mv_data = data;
rc = mdb_put(txn, dbi, &key, &data, 0 /*flags*/);
rc = mdb_txn_commit(txn);
```

To lookup data, one can either use `mdb_get(...)` or do so through a cursor. A cursor can also be used for put- and delete-operations, but this is not shown.

```c
MDB_cursor *cursor;
rc = mdb_txn_begin(env, NULL, MDB_RDONLY, &txn);
rc = mdb_cursor_open(txn, dbi, &cursor);
while ((rc = mdb_cursor_get(cursor, &key, &data, MDB_NEXT)) == 0) {
    printf("key: %p %.*s, data: %p %.*s\n",
        key.mv_data,
        (int) key.mv_size,
        (char *) key.mv_data,
        data.mv_data,
        (int) data.mv_size,
        (char *) data.mv_data);
}
mdb_cursor_close(cursor);
mdb_txn_abort(txn);
```

Finally, the database and environment is closed.

[5]http://symas.com/mdb/doc/

```
    mdb_close(env, dbi);
    mdb_env_close(env);
    return rc;
}
```

## 3.3 Architecture Overview

### 3.3.1 Introduction

LMDB stores data in a *copy-on-write B+tree*. B+trees have been used in database systems for decades, and is well-suited for efficient data storage and retrieval.(Comer, 1979) By implementing a copy-on-write variant of B+trees, LMDB is able to provide a form of multi-version concurrency control (MVCC), allowing write- and read-transactions to operate concurrently, without a complicated locking-scheme protecting readers from changes introduced by writers.

The rest of this section will contain the following: first a look at how the copy-on-write B+tree works, and how disk-pages are managed during copy-on-write operations. Then, a description of how writes and reads are performed on the database. Finally, issues regarding durability and recovery are covered.

### 3.3.2 Copy-on-Write B+Tree

Classical implementations of B+trees usually update pages in the tree in-place. This is very efficient, as the necessary changes are made directly in the tree, but creates issues when it comes to concurrent access by other transactions. Usually complicated locking-schemes are introduced so that other transactions are unable to access the change parts of the tree until the modifying transaction either commits or aborts.

LMDB avoids this kind of locking in favour of a multi-versioning system. The idea is to have write-transactions create a completely new version of the tree, leaving the old version intact, so that in-process read-transactions can—unobstructedly—continue to read the data from it until it is finished.

This is the origins of the copy-on-write approach used by LMDB. By never modifying parts of the tree in-place, but instead copying each page and then modifying this copy, a new version of the B+tree can be created by a write-transaction in complete isolation. If the transaction decides to commit, the new tree becomes the "current" tree—and thus the starting point for new read-transactions so that they will find the newly applied changes. Should the transaction instead abort, its changed tree is simply dropped meaning none of its changes ever take effect.

Figure 3.1 illustrated this approach. One can imagine a new data-item is to be placed in the rightmost leaf-page of the tree. Since this necessarily modifies the page, it is copied and the change is instead made to the copy. Because a copy was made, the root must be updated to point to this new page—requiring modification of the root itself. Therefore the root is also copied. In general, it is necessary to copy all pages on the path from the root to any modified page when creating a new version of the tree.

Once a new version of the tree is created and the creating transaction has committed, any new requests to the database can be directed to the new tree. However, since the old root

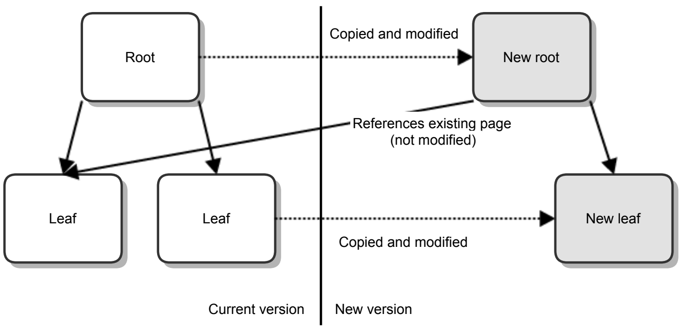

**Figure 3.1:** Copy-on-Write B+tree operations. Pages are copied on modifications, while any non-modified pages of the old version can be shared with the old version.

and its pages are still available and unmodified, existing read transactions can continue to access data in the old tree.

By handling all updates in this manner, LMDB archives its advertised ACID semantics. Since any changes does not become visible until a write-transaction commits, and the new tree becomes the current tree, *atomicity* of the changes is achieved. By maintaining any consistency-guarantees while creating the new tree, *consistency* is maintained. *Isolation* is trivial since only one write-transaction is active at a given time and all its changes are done to separate copies of pages. Finally, *durability* is guaranteed by making sure all changed pages are fully written to disk before the transaction is allowed to commit.

### 3.3.3 Page Management

Because each new version of the B+tree consumes a number of new pages (for new or copies of existing pages), it is important that there is also a system in place to *reclaim* pages once they are no longer needed.

The first consideration for such a system is to define *when* a given page is no longer used, so that it can be a candidate for re-use. In LMDB, old trees (and its pages) need to be available as long as they are referenced by any still-live transactions *or* the current version of the tree. Figure 3.2 illustrates this situation. The marked pages on the left side can be re-used once no live transactions reference this old version of the tree. The leftmost leaf can not be re-used however, since it is also referenced by the current version of the tree (right side of the figure).

There are many techniques once could use to identify such re-usable pages.[6] LMDB opts to record all pages freed by each write-transaction, and store this information in a sub-database. For the situation in figure 3.1, LMDB would record that—as part of the transaction—the old root and the old rightmost leaf was freed. (This is because they were

[6]This is essentially a garbage-collection problem.

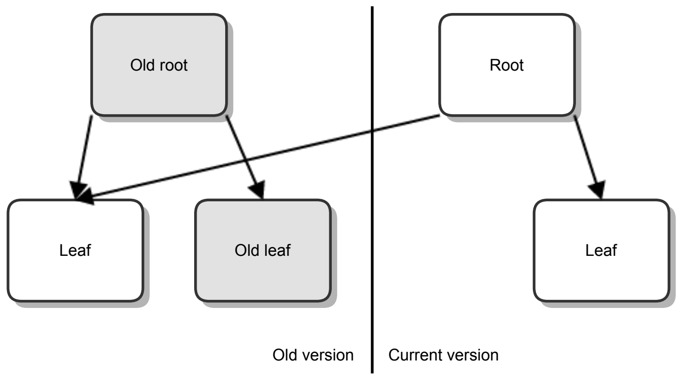

**Figure 3.2:** Identification of freed pages. Continuation of figure 3.1. Once the right-side version has become the current version, the marked pages on the left side can be freed once the old root is not referenced by any live transactions.

replaced by copies in the new tree.) The leftmost leaf was *not* recorded as freed, however, as no copy was made.

When a write transaction need a free page, the recorded free-page information can then be searched. By identifying the oldest version any active read-transaction is reading, all pages freed before or during the creation of that version are available for re-use. For example, if version 5 is the oldest version of the tree operated on by any read-transaction, and pages freed during the creation of versions 1, 2, 3, 4, and 5, is eligible. Any pages freed by the creation of version 6 would *not* be eligible, however. This is because such a page is necessarily referenced by the version-5 tree, which is currently in-use.

By checking for re-usable pages before possibly allocating new ones on-disk, disk-space usage can be kept to a minimum. This is a good policy, as once the database file reaches a given size it can not trivially be shrinked—as used pages will be fragmented through the file. This is also has consequences for page reads and writes, as the on-disk locations of pages can start to vary significantly.

### 3.3.4 Write Path

Once a write is to be performed, first a write-transaction[7] is started. This prompts LMDB to prepare the necessary information so that the current version of the B+tree can be read and a new version created. Once all changes have been made, the transaction is committed. This writes all changed pages to disk, records all pages freed during the transaction, and updates the database so that new transactions will work on the newly created tree.

The main challenges during this process are: finding pages to reuse, do operations on the

[7]More accurately, a read-write-transaction, as it can do both.

B+tree (adding and removing data, handling page-splits, updating page-pointers, and so on.), keeping track of freed pages, and to successfully patching the current state of the database to point to the new version of the database on commit.

### 3.3.5 Read Path

Reads during read-transactions does not involve much complexity, as once the read-transaction is started and set to point to the current version of the database—as well as registered as an active-reader, for page-reclamation reasons—lookups are generally a straightforward case of B+tree-lookups. Once the transaction completes, it unregisters itself from the list of active readers.

Write-transactions (or rather, non-read-only transactions) are not much more complicated, but care has to be taken so that when pages are copied and updated old pages aren't accidentally read instead of these newer copies.

### 3.3.6 Recovery and Durability

Durability in LMDB is achieved by flushing and syncing changed disk-pages on transaction commit. This means it is always more efficient to have many changes applied in *one* transaction, rather than many transactions with fewer changes, since this carries a significant cost—especially since not only changed leaf-pages, but all internal branch-pages on the root–leaf-path need to be written. Because of the page-reclamation technique used, the flushed pages are also unlikely to be located at sequential locations on disk, further increasing this cost.

On the other hand, recovery in LMDB extremely simple and straightforward as a consequence of the copy-on-write approach. Any failures during the creation or writing to disk of a new version is guaranteed to still leave the old version intact, meaning normal operation can resume immediately on startup—no special recovery steps are needed. The only requirement is that information about this "old" or "current" version is easily located on startup, and that replacing it with a new version is done in an atomic manner.

## 3.4 Implementation Highlights

### 3.4.1 Introduction

This section will look at some of the more interesting implementation details of LMDB, so that it is possible to gain a more detailed understanding of how it works.

While doing this, some of LMDB's more advanced features and toggleable behaviour are ignored. This is so focus can be kept squarely on how the basic key-value storage functionality is implemented without being overwhelmed by *special cases* used to support additional functionality or behavior. Some examples of features whose implementations are ignored are: nested transactions, multiple databases in a single file, multiple values for each key, and multi-process concurrency. Some toggleable behaviour which is ignored includes: the possibility of performing writes directly to the memory map, not syncing meta-pages, and allowing a single thread to have multiple active read-transactions by not using thread-local storage.

Below, details regarding LMDB's use of memory-mapping is covered. Then comes a look at how LMDB does free-page reclamation, the format of its on-disk pages, and how navigation through its copy-on-write B+tree is done. Finally implementation details in regards to supporting multi-versioning and recovery is covered.

### 3.4.2 Memory Mapped Data File

One of the initial goals of the LMDB project was to create a system where no buffer- and cache-configuration and management is necessary. LMDB achieves this by using the operation system's *memory-mapping* capability to access on-disk data, leaving caching and buffer-management up to the filesystem and OS instead of LMDB itself. This has a number of advantages:(Chu, 2012)

- No copying has to be done when returning data to the client: instead a pointer to the data's location in the memory mapped area can be returned. (Data integrity optionally can be protected by mapping the memory-area as read-only so the user does not accidentally change it.) This reduces the amount of work which has to be done in terms of memory allocation and copying before data is returned to the user, which has the potential to greatly speed up data-access.
- Data is not cached redundantly within the application in addition to the OS or file system caches. This reduces complexity inside LMDB, and also frees up memory with would otherwise be used for such an application-level cache. However, you rely on the OS' semantics regarding this.

For this approach to be viable, it forces a number of requirements and responsibilities on the operating system LMDB is running on. One is that the memory address space must be large enough to contain the expected data volume. For 32 bit systems, this sets a hard limit on 4 GB—but is likely to be less than this in practice. In recent years, 64 bit systems have become more common, which ups the limit to between hundreds of terabytes and multiple exabytes. This makes the scheme much more viable for more realistic database sizes. A second requirement is that the operating system and file system manages its buffers efficiently, else the approach becomes too slow to be usable in practice. Finally, when not doing writes to the memory map itself, LMDB requires that the operating system uses an unified buffer cache, meaning that changes to the data file through OS file-write calls is immediately visible through the map.(Chu, 2012)

Initially, LMDB did not handle *writes* by changing the memory map directly. Instead, they were performed using OS file-writing functions. This was so LMDB retained complete control of the ordering of writes to disk, when the write-back actually happen, as well as more easily being able to flush changes to disk. Eventually, support for memory map writing added, speeding up write-operations in the process. However, this carries a risk of corrupting the database through wild pointer writes and other bad updates, and requires that the operating systems provides some degree of control of as to when data is actually written back to disk.

### 3.4.3 Free Page Reclamation

When new pages need to be allocated, possibly because it is needed to make a copy of an existing page or because a page split, the function:

```
int mdb_page_alloc(MDB_cursor *mc, int num, MDB_page **mp)
```

. . . is called. The arguments are a cursor—identifying the current transaction and environment—the number of sequential pages to allocate, and a location to place the first of the ready-to-be-used pages (the remaining pages can be identified by looking at this page's page-number).

To locate pages eligible for re-use, the function first identifies the oldest currently running read-transaction by calling the function `mdb_find_oldest`. How this function works is covered in section 3.4.3 below.

Once the oldest running read-transaction's transaction ID is known, all pages freed by transactions older than it can safely be re-claimed. LMDB identifies such pages by storing all freed pages in internal database, where the freeing transaction's transaction ID is used as the key. By starting at the lowest transaction ID (key) in the free-page database, and looping until the ID is greater than or equal to the oldest transaction ID, all eligible pages are located. By keeping all eligible pages in a sorted list, a continuous range of pages can then be found. To logic for this search is as follows:

```
/* Seek a big enough contiguous page range. Prefer
* pages at the tail, just truncating the list.
 */
if (mop_len > n2) {
  i = mop_len;
  do {
```

```
 7       pgno = mop[i];
 8       if (mop[i-n2] == pgno+n2)
 9         goto search_done;
10     } while (--i > n2);
11     if (Max_retries < INT_MAX && --retry < 0)
12       break;
13   }
```

mop_len is the length of the mop (old-pages) list, and n2 is the number of contiguous pages requested minus 1. The mop list is sorted by decreasing page-numbers, so once the if on line 8 passes the range starting at page-number pgno and including pgno+n2 is found to be free.

If the request can not be satisfied by reclaimed pages, new pages are allocated by growing the data-file (as long as the maximum size is not exceeded):

```
/* Use new pages from the map when nothing suitable in the freeDB */
i = 0;
pgno = txn->mt_next_pgno;
if (pgno + num >= env->me_maxpg) {
    DPUTS("DB size maxed out");
    rc = MDB_MAP_FULL;
    goto fail;
}
[...]
txn->mt_next_pgno = pgno + num;
```

When a write-transaction is committed or aborted, the necessary changes to the free-page database is applied and committed and aborted with it. This keeps the register of free-pages consistent.

### Active Readers Table

LMDB maintains a table of active read-transactions. The first time a thread starts a read-transaction, it is allocated a slot in this table. This requires a mutex to protect the table from concurrent changes:

```
LOCK_MUTEX_R(env);
nr = ti->mti_numreaders;
for (i=0; i<nr; i++)
  if (ti->mti_readers[i].mr_pid == 0)
    break;
if (i == env->me_maxreaders) {
  UNLOCK_MUTEX_R(env);
  return MDB_READERS_FULL;
}
ti->mti_readers[i].mr_pid = pid;
ti->mti_readers[i].mr_tid = tid;
if (i == nr)
  ti->mti_numreaders = ++nr;
/* Save numreaders for un-mutexed mdb_env_close() */
env->me_numreaders = nr;
UNLOCK_MUTEX_R(env);
```

A slot with `mr_pid == 0` is free (line 4). Once a free slot is found, this is stored in thread-local storage so that the slot can be re-used as long as the thread is live (last part of line 3):[8]

```
1  r = &ti->mti_readers[i];
2  new_notls = (env->me_flags & MDB_NOTLS);
3  if (!new_notls && (rc=pthread_setspecific(env->me_txkey, r))) {
4    r->mr_pid = 0;
5    return rc;
6  }
```

Once the thread exits, the slot will be freed:

```
[...]
pthread_key_create(&env->me_txkey, mdb_env_reader_dest);
[...]

static void
mdb_env_reader_dest(void *ptr)
{
  MDB_reader *reader = ptr;
  reader->mr_pid = 0;
}
```

By acquiring a slot once, instead for example at the start of every transaction, lock-contention between threads performing reads is minimized.

With this structure in place, the oldest read transaction can be located as follows:

```
static txnid_t
mdb_find_oldest(MDB_txn *txn)
{
  int i;
  txnid_t mr, oldest = txn->mt_txnid - 1;
  if (txn->mt_env->me_txns) {
    MDB_reader *r = txn->mt_env->me_txns->mti_readers;
    for (i = txn->mt_env->me_txns->mti_numreaders; --i >= 0; ) {
      if (r[i].mr_pid) {
        mr = r[i].mr_txnid;
        if (oldest > mr)
          oldest = mr;
      }
    }
  }
  return oldest;
}
```

No locks or other synchronization is used here, so read-transactions will never block write-transactions. A consequence of this is that somewhat stale data can be returned, however this matters little in practice.

[8]While the code shows pthread-specific calls, these are actually `#define`ed to OS-specific functions on non-POSIX systems, such as Windows.

### 3.4.4 Page Layout

When storing pages in the data-file, the format illustrated in figure 3.3 is used. A header is included on every page (except for overflow-pages, where only the first page in such a sequence has a header), and the rest is used according to the type of page—examples of which include meta-pages, branch pages, and leaf pages.

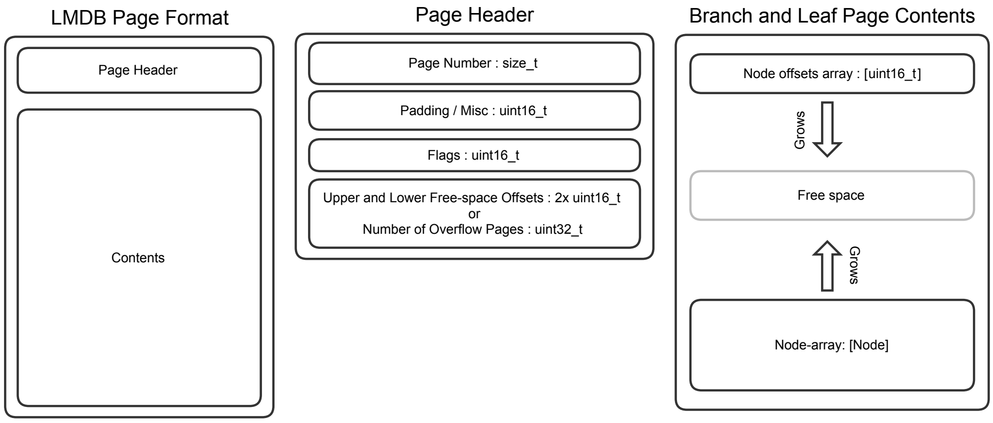
LMDB Page Format
Page Header
Contents
Page Header
Page Number : size_t
Padding / Misc : uint16_t
Flags : uint16_t
Upper and Lower Free-space Offsets : 2x uint16_t
or
Number of Overflow Pages : uint32_t
Branch and Leaf Page Contents
Node offsets array : [uint16_t]
Grows
Free space
Grows
Node-array: [Node]

**Figure 3.3:** LMDB's disk-page format. A page has a header of fixed size, and the rest is filled with page-type specific contents. The header includes the page-number of the page (location where it belongs in the data-file), flags indicating the contents of the page, as well as information about available free-space (or if the page is the first page of a sequence of overflow-pages, the number of such pages are stored in these bits). For leaf and branch-pages, the contents of the page is a number of nodes.

For branch and leaf pages, the remaining space after the header is used to store a number of nodes. Each node is used to hold a single key-value pair, which either stores downwards tree-pointers (in branch-pages) or user-data (in leaf-pages). Immediately after the header, an array of `uint16_ts` is placed denoting offset of each stored node—in order sorted by the node-key's value. This array grows downwards in the file as new nodes are added. After this list comes all free space in the file, before the actual array of nodes starts. This array consequently grows upwards in the file—towards the free space. To calculate if a page has the necessary available space for a new node, the sum of a node-offset-pointer and the node itself is calculated and compared to the available free space. If a node is removed, the node-offset-pointer is removed and all nodes after it moved to fill the freed space, and the same is done for the actual node data. All free space is therefore always located in the middle of the file.

To track where the free space starts and ends (alternatively, where the node-offset-array ends and later where the node-array begins), the page-header stores offsets into the file to where these cutoffs are located (upper and lower free-space offsets). By storing node-offsets first, and in sorted-by-key order, this list can be binary-searched to quickly locate a node with a given key or insertion point in the offsets-list for a new node.

The nodes themselves has the format illustrated in figure 3.4. First, two `unsigned shorts` are stored, which together denotes the size of the key-value pair's value (for leaf-pages) or the page-number the node points to (for branch-page). (It is stored as two `unsigned shorts` for micro-optimization reasons.) Next is a field used to store additional information about the node (used for some of LMDB's advanced features, for example keys with multiple values), then the length of the key-value pair's key. Finally comes a variable-length array used to store the actual key- and value-data..

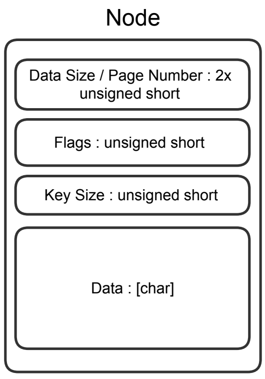
Node
Data Size / Page Number : 2x unsigned short
Flags : unsigned short
Key Size : unsigned short
Data : [char]

**Figure 3.4:** Format of a single node, used to store key-value data in branch- and leaf-nodes.

To access a given node's key and value, the following C macros can therefore defined:

```
/** Address of the key for the node */
#define NODEKEY(node)   (void *)((node)->mn_data)
/** The size of a key in a node */
#define NODEKSZ(node)    ((node)->mn_ksize)
/** Address of the data for a node */
#define NODEDATA(node)   (void *)((char *)(node)->mn_data + (node)->mn_ksize)
/** Get the size of the data in a leaf node */
#define NODEDSZ(node)    ((node)->mn_lo | ((unsigned)(node)->mn_hi << 16))
```

Given all this, a page can be binary-searched for a node with a specified key in this way:

```
    while (low <= high) {
      i = (low + high) >> 1;

      node = NODEPTR(mp, i);
      nodekey.mv_size = NODEKSZ(node);
      nodekey.mv_data = NODEKEY(node);

      rc = cmp(key, &nodekey);

      if (rc == 0)
        break;
      if (rc > 0)
        low = i + 1;
      else
        high = i - 1;
    }
```

…where `cmp(...)` is the (possibly client-specified) key-comparator function, used to determine ordering among different keys, and `NODEPTR(page, i)` uses the node-offsets-array to calculate the location of node `i` in the page.

By using this kind of structure, LMDB is able to quickly locate where a key-value pair is stored in a page, as well as to add a new key-value pair without having to move around a large amount of data (only the node-offsets-list need to be changed). Deletions are the most costly operation, as first the hole left by the node is filled by moving nodes above it in the file, and then the offsets of these nodes are updated.

### 3.4.5 B+Tree Navigation

When using a B+tree, it is necessary to have some way to navigate through it. The interior nodes always include downwards pointers, but it is also often necessary to perform upwards or sideways traversal. Examples of sideways navigation is to locate the left or right sibling of a leaf-node when performing a range-scan of the stored data, or to locate a leaf-page's parent when splitting a page.

Including such pointers in the nodes themselves introduce additional layers of complexity in order to maintain them. For a copy-on-write B+tree approach, such as LMDB's, it is additionally not even a viable solution to store neither sibling-pointers nor parent-pointers. To see why this is so, consider that the root-page always have to be changed when any leaf is changed.[9] This necessitates update of *all* its child-pages so that they point to this new root-location. Recursively all pages in the tree need to be updated as well—resulting in the entire tree being copied. The same reasoning goes for sibling-pointers, as the change of a leaf required all its siblings to be updated (as well as that sibling's parent)—further requiring *their* siblings (and parents) to be updated. Storing such pointers is therefore simply not a viable option for a copy-on-write B+tree.

To handle B+tree traversal, LMDB instead maintains a data-structure to keep track of its location in the tree at run-time. The `MDB_cursor` struct includes a stack where page-pointers are pushed as the cursor descends from the root of the tree.

```
#define CURSOR_STACK       32

struct MDB_cursor {
   [...]
   unsigned short  mc_snum;/**< number of pushed pages */
   unsigned short  mc_top; /**< index of top page, normally mc_snum-1 */
   [...]
   MDB_page *mc_pg[CURSOR_STACK]; /**< stack of pushed pages */
   indx_t    mc_ki[CURSOR_STACK]; /**< stack of page indices */
};
```

The page a cursor is located at will therefor be `cursor.mc_pg[cursor.mc_top]`, and the parent of that page will be `cursor.mc_pg[cursor.mc_top-1]`. Since it is also often necessary to know which pointer on a branch-page was followed (to be able to

[9]Because the leaf's parent, and recursively this page's parent, need to be updated to point to the new version.

identify siblings), the on-page index of followed pointers are stored in the `mc_ki` stack. A page's right sibling can thus be located by following the `mc_ki[m_top]+1`th pointer (if it exists) in the parent page.

### 3.4.6 Multi-Versioning and Recovery

Instead of explicitly maintaining *all* previous versions of the database, LMDB instead only preserves the last two—the *current* and *previous* version. These are each represented by a *meta page*, located at respectively page number 0 and 1 in the database file. A committing write-transaction will have started by reading the current version's meta-page, and will overwrite the previous version's meta-page with a new page representing the new state of the database. This new version then becomes the current version to which new transactions are directed. A meta-page includes information such as the last allocated page in the data-file, transaction ID of the transaction writing the meta-file, free-page- and main-database information (for example their root pages), and size of the memory-mapped region.

By maintaining this information in a fixed location on disk, and synchronously flushing changes to it before each transaction is deemed to have successfully committed, recovery becomes extremely simple: read the two meta-pages, and compare their transaction ID to identify the newest version. Once this is done, normal operations can resume with no additional recovery steps. The reason this works is because the writing of the meta-page is the final step of the commit-process; all other changes are guaranteed to have been successfully written to disk. If anything goes wrong before the meta-page is written, recovery works because it is the *last* version that is being overwritten—the *current* version will remain unchanged and intact. Once a new meta-page has been successfully written (overwriting the *last* version), this version will persist until it is successfully replaced.

Implementing this simply means writing the meta-page *last* during transaction commit. This is handled by `mdb_txn_commit`:

```
if ((rc = mdb_page_flush(txn, 0)) ||
    (rc = mdb_env_sync(env, 0)) ||
    (rc = mdb_env_write_meta(txn)))
    goto fail;
```

. . . which first writes then flushes all changed pages to disk, before then writing and flushing the meta-page. On startup, as part of `mdb_env_open(...)`, `mdb_env_read_header(...)` is called which reads the two stored meta-pages and returns the most recent meta-page (the one with the highest stored transaction ID).

By only explicitly maintaining two versions, this still leaves the issue of how to retain the needed information about older versions needed by long-running read-transactions. (Starting a read-transaction, and then committing two write-transactions will necessarily overwrite information that was current when the read-transaction started.) LMDB solves this by having transactions copy database-information from the current meta-page on transaction start (from `mdb_txn_renew0(...)`):

```
// Copy the DB info and flags
memcpy(txn->mt_dbs, meta->mm_dbs, 2 * sizeof(MDB_db));
```

The database-information (`MDB_db`) includes data such as the location of B+tree root pages, so that a search can be started from it. With this information always available, the remaining necessary condition is that the pages this version references are never overwritten by newer pages—as was the subject of section 3.4.3.

## 3.5 Discussion

### Recovery Method

This recovery mechanism of LMDB is reminiscent of IBM System R's *"shadowed, no-log"* file option, where the content of a file after a crash is whatever a *shadow file* contains.[10] As noted by the System R developers, this works well as long as only a single write-transaction is in-progress at a given time, but they were unable to *"architect a transaction mechanism based solely on shadows which supported multiple users and save points"*.(Gray et al., 1981) Instead, they found it necessary to implement logging of UNDO and REDO records to support this—at which point the shadows themselves become redundant, since one is essentially doing write-ahead-logging. Since LMDB abandons the support for multiple concurrent write-transactions, however, it is is able to use this shadow-page technique to provide a very simple, fast, and robust recovery mechanism—though at the cost of forgoing in-place modification for a copy-on-write approach.

When using this technique, committing a write-transactions requires writing all changed leafs as well as all pages on the path from the root to these leaves—plus the meta-page. A single changed leaf therefore require a number of disk IO-operations equal to the height of the B+tree plus one. These writes, at non-sequential locations on disk, has to be performed and then synchronously flushed to disk before the commit is finalized. This can be contrasted to write-ahead-logging systems where only the log has to be flushed before a transaction can commit.

### Page Fragmentation

A consequence of LMDB's page reclamation technique is that the location of logically sequential pages in the leaves of the B+tree is rarely logically sequential on disk. This means a disk seek if often necessary to locate B+tree sibling pages in a range query. This issue was also noted in System R's use of shadow pages: *"When a page is updated for the first time, it moves. Unless one is careful, and the RSS[11] is not careful about this, getting the next page frequently involves a disk seek"*.(Gray et al., 1981) It is possible to mitigate this by making the effort to locate *adjacent* free pages, however LMDB takes the stance that this fragmentation is an acceptable tradeoff compared to the effort of doing this.(Chu, 2013a)

### Copy-on-Write B+Tree

A variant of LMDB's copy-on-write B+tree (using free-page reclamation) is to do copy-on-write but never actually re-use old pages—instead only *appending* new ones to the back

[10]A shadow file is a copy of the file taken before modifications were done to it—in other words, the old, pre-modifications, version.

[11]System R's Research Storage System.

of the database file.[12] One database which implements this append-only variant is Apache CouchDB.(J. Chris Anderson, 2013)

This approach will give very different performance characteristics because it is allowed to *always write sequential pages on disk.* (Compare this to LMDB, where each page-write potentially involves a disk-seek.) However, because the file is only ever growing (new pages are always appended), there is likely going to be a need to reclaim old pages at some point—or face running out of disk space.[13]

CouchDB facilitates this by running a *compaction process*, which goes through the database file and identifies and deletes unneeded pages.(CouchDB, 2014) This can either be requested manually, or be automatically triggered based on various criteria. A consequence of this, however, is that once the compaction process is running database performance is likely to be impacted from the additional load. This, along with the additional complexities of implementing such a process, is one of the cited reasons for why LMDB wanted to avoid it—instead opting for its free-page management approach and accepting the random writes as a consequence.(Chu, 2013b)

## Single Level Storage

As covered in section 3.4.2, LMDB uses memory-mapping functionality to avoid having to deal with application-level caching and configuration challenges related to this. They attribute this idea to the concept of a *single-level store*.(Gray, 1993) While the single-level storage-concept usually refers to having *all* available storage on a computer is in a single address space, LMDB use it to refer to how the entire database is accessed through this single memory-mapped area.(Chu, 2012) While this is an old concept, it has recently become more viable because of the availability of computers with larger memory address space. As has always been the case, it can work well but depends heavily on how well the underlying systems handles maintaining the abstraction. In LMDB's case, it also does not allow for additional optimizations where LMDB can take advantage of the additional information it has when it comes to expected access patterns.

[12]The recovery mechanism for an append-only B+tree will be a little different from what LMDB does: instead of locating meta-pages at fixed locations in the file, a scan backwards from the end of the file is done until the first meta-page/root-page is located.

[13]LMDB cite how their initial attempt at a append-only design had a test-run grow the DB-file to 1027 pages, where only 10 pages containing current data—i.e. 99% of the total space "wasted".(Chu, 2012)

Chapter 4

# Evaluation

## 4.1 Introduction

The sections in this chapter will compare LevelDB and LMDB, first in terms of features, and then performance, to give a sense of in which scenarios one might be appropriate over the other.

## 4.2 Feature Comparison

Both systems have a very similar featureset:

- Both support get, put and delete operations.
- Both support sorted iteration of entries.
- Both support arbitrary byte-arrays as both key and value, although LMDB has a maximum key-length (512).
- Both support custom key-comparators, so that the user is able to define a custom order among keys.
- Both allow for multithreaded concurrency: a single concurrent writer for both (automatically enforced), and N (128 by default) readers for LMDB, while LevelDB has no limit on readers.

The main difference in terms of features is LMDB's support for transactions. LevelDB does not have this, but does offer a more limited alternative in the form of atomic batches.

This does not help—for example—in the event a user want to safely (in terms of maintaining database consistency) read a value, and then make a change based on this value, however, so if this is required LMDB is the obvious choice.

LMDB also offers additional features, such as multi-valued keys, and sub-databases. The latter means changes to multiple named sub-databases can be made during a single transaction. Users should therefore consider is such features are useful to them, and consider LMDB if so.

LevelDB makes it possible to create and delete transient snapshots, so that these can be stored and shared at runtime. In LMDB, a read-transaction works on a snapshot of the database, but once it closes the snapshot is gone. It is conceivable for a user to want to manually create a snapshot so that this can be referred back to over time, and is only provided by LevelDB. Snapshot-lifetimes being limited to the application run-time (meaning they are gone after a crash or application restart) lowers the general usefulness of this feature, however.

LevelDB can also automatically compress the database contents, something not done by LMDB. LevelDB was additionally found to have more efficient disk-space usage, even without compression enabled. So if disk-space usage is at a premium LevelDB is likely the preferred choice.(Symas, 2012)

## 4.3 Performance

This section will present the results of microbenchmarks to illustrate the relative performance of LevelDB and LMDB. First, two consequences of their different architecture with regards to write-latency is presented. Next, more comprehensive microbenchmarks are presented to show how the two systems compare more in general.

### Cost of a Single Write

It is instructive to compare the overhead involved in a single synchronous put-operation. For LevelDB, a single write—while there is space in the memtable—involves adding it to the memtable, as well as appending it to the on-disk log file. For LMDB, a single transaction-wrapped put-operation will place the entry in the correct leaf, then synchronously write this leaf and all pages on the root–leaf-path, as well as the meta-page to disk.

Figure 4.1 shows the relative costs of these operations. It displays the time taken by each put-operation after starting from an empty database. The key-value pairs consists of a 4-byte key and 1-byte value so that the majority of time spent will be overhead of adding the entry, not writing the entry itself.

For comparison, LevelDB's asynchronous mode is also shown. It does not do, or wait on, any IO as long as there is space in the memtable, so each put-operation is extremely

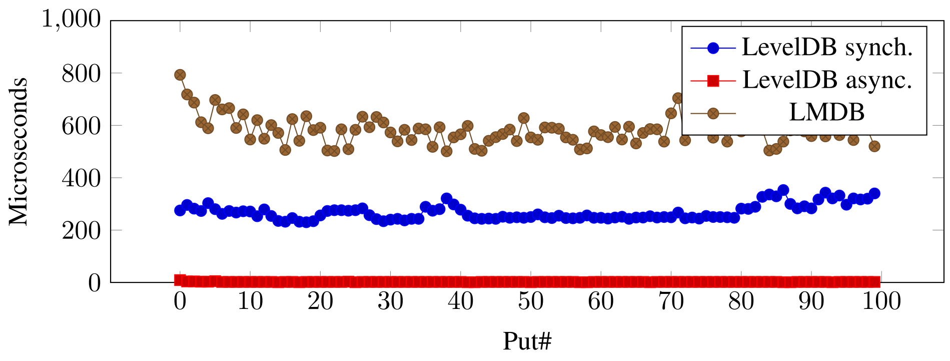

**Figure 4.1:** The duration of `Put`-operations with a 4-byte key and 1 byte value. No compactions were running or started for the duration.

cheap—only taking 2 microseconds on average. When turning on synchronous writes, however, LevelDB is forced to append the written entry to the log-file. This increases the duration to around 280 microseconds on average. LMDB takes around twice as long as LevelDB for each put-operation, averaging around 590. This makes sense, as it has to write at least 2 pages—the changed leaf as well as the meta-page.[1]

## Writes Over Time

LevelDB's architecture allows it to handle most of the incoming writes by appending it to the memtable and log. However, once the memtable fills up and is replaced (and scheduled for compaction), a single write can expect to see a significant spike in latency.

This is illustrated in figure 4.2, where both systems have key-value pairs consisting of a 4 bytes key and 256 KB value added to them over time. While LMDB consistently takes around 3300 microseconds to handle each transaction, LevelDB is significantly faster (around 500 microseconds) most of the time, but sees a spike after each 4 MB of written data, as the memtable is replaced—at which point a single write takes around 77 milliseconds to complete. This is an example of the varying write-latency of LevelDB as compactions are triggered, and projects extending LevelDB often introduce changes aimed at remove or reducing these kinds of spikes (refer to section 2.5).

## Microbenchmarks

Symas has produced a set of microbenchmarks comparing LMDB and LevelDB, as well as some other systems not covered here.(Symas, 2012) The benchmark code is based on

[1]However, it is worth noting how these durations are *extremely* short—much shorter than the expected time for data to actually hit the hard drive platter. The likely culprits are caches located between the drive and file system, so the durations themselves should not be given much weight. As an illustration of the relative difference between the two systems' approach, the comparison can still be said to have some value however.

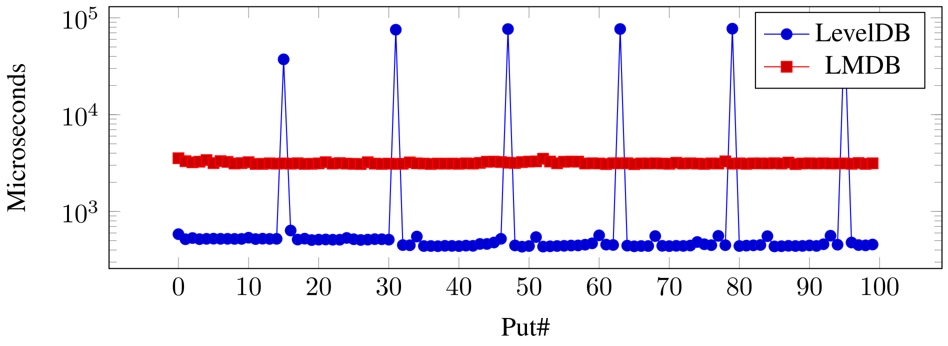

**Figure 4.2:** The duration of each `Put`-operation, starting from an empty database, with 4-byte keys and 256 KB values. At regular interval, after about 4 MB of data written, LevelDB incurs a stall (taking around 77 milliseconds—note how this is a log-lin plot) as memtable compaction is initiated. Each non-stalled write takes about 500 microseconds. LMDB consistently stays at around 3300 microsecond.

Google's own code released with LevelDB in 2011, but updated to fixed problems with regards to random-number generation.

The set of benchmarks is very comprehensive, and test the systems on multiple file systems as well as with different workloads. Examples include an in-memory file system, a 7200 rpm hard drive, an SSD, and some of the configurations are sequential and random reads and writes, batched writes, with large or small key-value pairs, using additional memory for caching, and in synchronous or asynchronous write-modes.

Below, some of the results for reading and writing to a 7200 rpm notebook hard drive, running the EXT2 file system, are presented. This is chosen because it can be considered reasonably representative for the kinds of systems the average user is likely to run LevelDB or LMDB on. They are also fairly illustrative of the workloads each system generally performs well on. Except for the synchronous sequential writes benchmark, where the magnitude was different but relative performance-difference the same, results are more-or-less identical to what was seen on an SSD.

### Reads

Figure 4.3 (sequential) and figure 4.4 (random) shows the results of reading key-value pairs consisting of 16 byte keys and 100 byte values. LMDB has a significant advantage in this area, and this can be traced back to how it is able to return data to the caller without first having to create a separate copy. Increasing the value-size (not shown) only magnifies LMDB's advantage, showing how copying quickly ends up dominating the runtime in such cases.

Random reads are generally much more expensive, and LMDB maintains it advantage here as well. LevelDB takes the highest impact compared to sequential reads; this makes sense

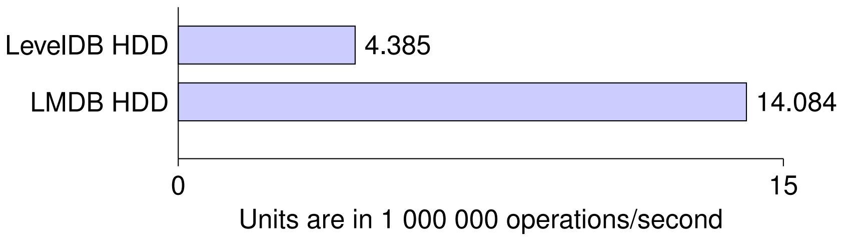

**Figure 4.3:** Sequential reading of 16B/100B key-value pairs.

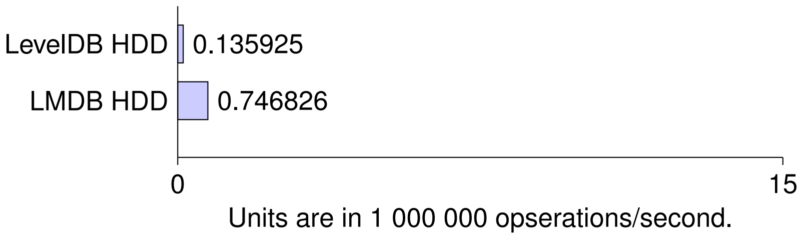

**Figure 4.4:** Random reading of 16B/100B key-value pairs

as random reads are likely to have to be served from the higher-numbered levels (where the majority of the data is located), which are never checked before both the memtables and all lower-numbered levels have been examined first. LMDB's B+tree does not suffer from similar issues.

### Writes

Figure 4.5 shows the results for synchronous sequential writes of 16+100 byte key-value pairs. The results for random writes (not shown) are similar for LevelDB, but sees LMDB drop by around 18

Here LevelDB's LSM-style design comes into its own, as it is able to service each write using only a single synchronous IO-operation—writing the log—and defers (and batches) additional disk-writes for later. LMDB's design, requiring at least least two pages (at different locations) to be written, means the cost of synchronous writes has a large impact on overall performance.

By increasing the value-sizes (figure 4.6), however, the gap closes. A probable reason for this is that, with larger values, the memtable fills up more quickly, and the LSM-style design becomes a burden as each key-value pair has to be written twice in relatively quick succession (first to the log, then during memtable compaction)—wiping out the advantages otherwise provided by the defer-and-batch design when faced with smaller values. LMDB only write each key-value pair once, so it only incurs the additional cost of having to transfer somewhat larger values, but otherwise does largely the same amount of processing.

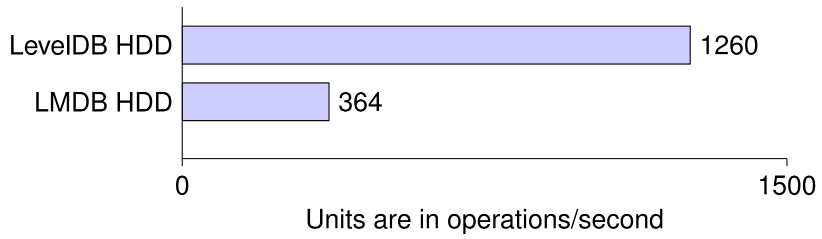

**Figure 4.5:** Synchronous sequential writes of 16 + 100 byte key-value pairs.

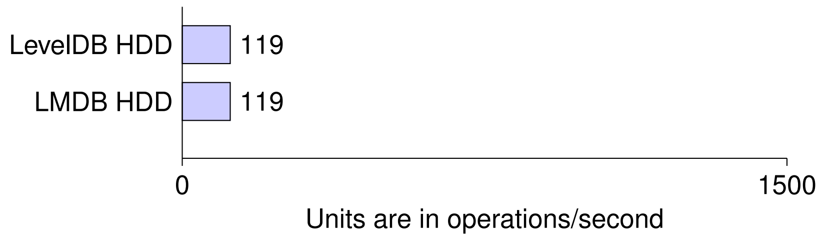

**Figure 4.6:** Synchronous sequential writes of 16 + 100 000 byte key-value pairs.

Figure 4.7 shows what happens if synchronous writes are disabled, meaning the potential of data-loss is introduced should the system crash. (Note that the values in the figure are in 1000 ops./sec., not ops./sec.) Compared to figure 4.5, LevelDB's advantage has essentially disappeared. This is to be expected however, as the goal of the LSM-style design is to work around the relatively high cost of synchronous IO-writes. Taking this cost out of the equation entirely means alternative designs, such as LMDB's is likely to be preferable. Indeed, doing random writes in this situation (figure 4.8) sees LMDB pull ahead.

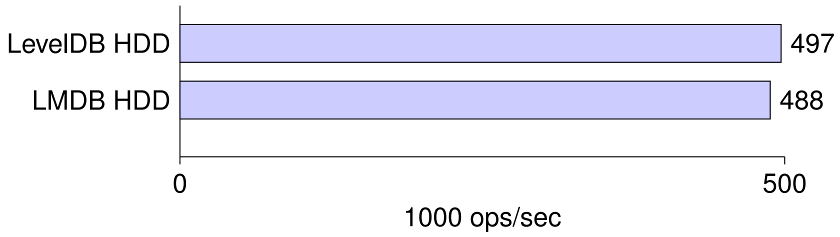

**Figure 4.7:** Asynchronous sequential writes of 16 + 100 byte key-value pairs.

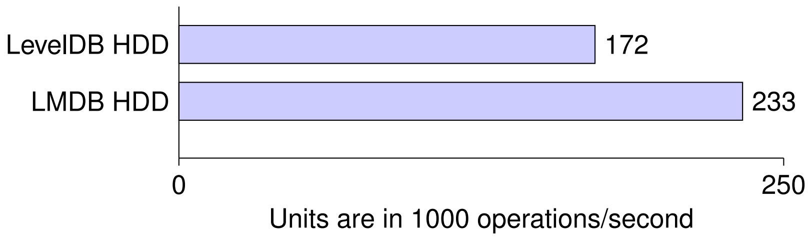

**Figure 4.8:** Asynchronous random writes of 16 + 100 byte key-value pairs.

## 4.4 Conclusions

In conclusion, LevelDB and LMDB both solves the general problem of key-value storage, but does so in very different ways.

LMDB, through its use of a memory-map and copy-on-write B+tree, emphasises read-performance—as was the initial goal—and is far superior to LevelDB in this regard. LevelDB, on the other hand, is shown—through its use of memtables, sstables, and compactions—to lay a groundwork allowing for a large degree of sequential, rather than random, disk IO operations. This is seen to pay off when writing data as long as entries are not too large—at which point the cost of facilitating this appears to outweigh the benefits.

On the LMDB website, the following is noted about its performance:

> *It is a read-optimized design and performs reads several times faster than other DB engines, several orders of magnitude faster in many cases. It is not a write-optimized design; for heavy random write workloads other DB designs may be more suitable.(Symas, 2014)*

Google does not make any similar statements about LevelDB, but the LSM-inspired design informs its priorities. The abstract of the LSM-paper notes the following:

> *The Log-Structured Merge-tree (LSM-tree) is a disk-based data structure designed to provide low-cost indexing for a file experiencing a high rate of record inserts (and deletes) over an extended period. [...] the LSM-tree is most useful in applications where index inserts are more common than finds that retrieve the entries. This seems to be a common property for History tables and log files, for example.*(O'Neil et al., 1996)

As such, a user looking to chose between LevelDB and LMDB should carefully consider both which features are needed, and the kind of workload is likely to be presented to the storage system. As with most areas in computer science, there is unlikely to exist any kind of *silver bullet* appropriate for all use cases, so the different tradeoffs presented by each system have to be considered depending on the exact situation at hand.

### Future Work

While LMDB uses a B+tree in its implementation, and LevelDB employs very similar structure to LSM-trees, this thesis does not look at any *hashing based* systems. Examining such a system, and comparing both its implementation and performance characteristics to these alternative approaches, would therefore be an interesting exercise. An example of such an embedded key-value store is *Kyoto Cabinet*.[2]

[2] http://fallabs.com/kyotocabinet/

# Bibliography

Anderson, A., 1989. Optimal bounds on the dictionary problem. In: Proceedings of the International Symposium on Optimal Algorithms. Springer-Verlag New York, Inc., New York, NY, USA, pp. 106–114.
URL http://dl.acm.org/citation.cfm?id=91896.91913

Apache, 2014. Apache activemq – leveldb store.
URL http://activemq.apache.org/leveldb-store.html

Basho, 2014a. Basho technologies: Leveldb.
URL http://docs.basho.com/riak/latest/ops/advanced/backends/leveldb/

Basho, 2014b. Riak docs – backends: Leveldb.
URL http://docs.basho.com/riak/latest/ops/advanced/backends/leveldb/

Bitcoin, 2013. Bitcoin 0.8 release notes.
URL https://github.com/bitcoin/bitcoin/blob/v0.8.0/doc/release-notes.txt

Burd, G., January 2011. Is berkeley db a nosql solution?
URL https://blogs.oracle.com/berkeleydb/entry/is_berkeley_db_a_nosql_solutio

Cattell, R., 2011. Scalable sql and nosql data stores.
URL http://cattell.net/datastores/Datastores.pdf

Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., Gruber, R. E., 2006. Bigtable: A distributed storage system for structured data. In: IN PROCEEDINGS OF THE 7TH CONFERENCE ON USENIX SYMPOSIUM ON OPERATING SYSTEMS DESIGN AND IMPLEMENTATION - VOLUME 7. pp. 205–218.

Chu, H., 2012. Life after berkeleydb: Openldap's memory-mapped database.
URL http://symas.com/mdb/20120829-LinuxCon-MDB-txt.pdf

Chu, H., August 2013a. Is lmdb a leveldb killer?
URL https://symas.com/is-lmdb-a-leveldb-killer/

Chu, H., April 2013b. The lightning memory-mapped database (lmdb).
URL http://symas.com/mdb/20130406-LOADays-LMDB.pdf

Comer, D., 1979. The ubiquitous b-tree. ACM Computing Surveys 11, 121–137.

CouchDB, 2014. Couchdb wiki: Compaction.
URL https://wiki.apache.org/couchdb/Compaction

Ellis, J., 2011. Leveled compaction in apache cassandra.
URL http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra

Fraser, K., 2004. Practical lock-freedom. Ph.D. thesis, University of Cambridge.
URL http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf

Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., Traiger, I., Jun. 1981. The recovery manager of the system r database manager. ACM Comput. Surv. 13 (2), 223–242.
URL http://doi.acm.org/10.1145/356842.356847

Gray, R., 1993. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers.

Hobbs, T., 2012. When to use leveled compaction.
URL http://www.datastax.com/dev/blog/when-to-use-leveled-compaction

HyperDex, 2013. Hyperleveldb performance benchmarks.
URL http://hyperdex.org/performance/leveldb/

HyperDex, 2014. Hyperdex homepage.
URL http://hyperdex.org

J. Chris Anderson, Jan Lehnardt, N. S., 2013. Couchdb: The definitive guide.
URL http://guide.couchdb.org/editions/1/en/btree.html

Jeff Dean, Sanjay Ghemawat, S. T., 07 2011. Leveldb: A fast persistent key-value store.
URL http://google-opensource.blogspot.no/2011/07/leveldb-fast-persistent-key-value-store.html

Jin, L., 2014. Rocksdb architecture guide.
URL https://github.com/facebook/rocksdb/wiki/Rocksdb-Architecture-Guide

Knuth, D. E., 1973. The Art of Computer Programming, Volume III: Sorting and Searching. Addison-Wesley.

LevelDB, 2012. Leveldb documentation.
URL `http://leveldb.googlecode.com/svn/trunk/doc/index.html`

LevelDB, 2014. Leveldb - google project hosting.
URL `https://code.google.com/p/leveldb/`

Maurice Herlihy, Yossi Lev, V. L. N. S., 2012. A provably correct scalable concurrent skip list.

Michael A. Olson, Keith Bostic, M. S., 1999. Berkeley db. Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference.
URL `https://www.usenix.org/legacy/events/usenix99/full_papers/olson/olson.pdf`

O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E., Jun. 1996. The log-structured merge-tree (lsm-tree). Acta Inf. 33 (4), 351–385.
URL `http://dx.doi.org/10.1007/s002360050048`

Poznyakoff, S., 2011. Gnu dbm.
URL `http://www.gnu.org.ua/software/gdbm/`

Pugh, W., Jun. 1990. Skip lists: A probabilistic alternative to balanced trees. Commun. ACM 33 (6), 668–676.
URL `http://doi.acm.org/10.1145/78973.78977`

Symas, September 2012. Database microbenchmarks.
URL `http://symas.com/mdb/microbench/`

Symas, 2014. Symas lightning memory-mapped database (lmdb).
URL `http://symas.com/mdb/`

Walsh, L., September 2011. Write optimization: Myths, comparison, clarifications.
URL `http://www.tokutek.com/2011/09/write-optimization-myths-comparison`