University of Applied Sciences Albstadt-Sigmaringen
Master's Program in Digital Forensics

Master's Thesis

# Development of a Forensic Parser for LevelDB

*(Translation of: Entwicklung eines forensischen Parsers für LevelDB, 2025)*

January 24, 2026

Alexander Hübert
E-Mail: alexander.huebert@protonmail.com

Supervisor:
Prof. Holger Morgenstern
David Schlichtenberger, M.Sc.

*Verum esse ipsum factum.*

– Giambattista Vico, *De Antiquissima Italorum Sapientia* (1710)

**Abstract**

This thesis examines the structure and layout of LevelDB and, based on this foundation, presents the development of a parser capable of extracting and processing the available information. Building upon a custom-developed library, two applications were implemented to address complementary requirements. A GUI-based parser enables a clear visualization and fast searching of the textual content of multiple databases simultaneously, while a CLI-based parser allows for a detailed analysis of individual database files, including their metadata. Together, these tools provide both an efficient overview and an in-depth examination of LevelDB databases.

As part of a subsequent validation phase, the parser is subjected to various functional and performance tests and demonstrated through a practical example involving the analysis of the Microsoft Teams application.

In addition, a dedicated section illustrates the use of LevelDB in Chromium-based applications, particularly in connection with the web APIs *Web Storage* and *IndexedDB*, to highlight the database's role in common and practical scenarios.

# Table of Contents

# Chapter 1

# Introduction

The idea to develop a forensic parser for LevelDB arose during the *Browser and Application Forensics* module led by Professor Morgenstern. As part of this module, the Android application Notion was examined[1], which uses LevelDB in addition to SQLite for the persistence of application data.

A search for a freely available, standalone forensic parser for extracting and analyzing the data generated from LevelDB was unsuccessful. This gap will be closed by developing such an application as part of this thesis.

In the course of this thesis, the structure and design of the LevelDB database will be analyzed and described in order to implement a standalone forensic parser based on the findings.

This chapter will first outline a rough overview of the scope of application of LevelDB, followed by a description of the underlying problem and the objectives of this thesis. The state of research then provides an overview of relevant technical work on the topic, before finally discussing in more detail the intended approach to the development and implementation of the planned forensic application for the LevelDB database.

## 1.1 Application Area of LevelDB

LevelDB is an open-source, lightweight database that has been developed by Sanjay Ghemawat and Jeff Dean at Google since 2011. Like the widely used SQLite database, LevelDB can be embedded directly and resource-efficiently into a local or mobile application, for example, without relying on a separate database server. Unlike the relational database SQLite, LevelDB belongs to the group of so-called

NoSQL databases. Specifically, it is a key-value database that stores and retrieves data as simple key-value pairs without predefined table structures and therefore does not use SQL[1] or relational structures. Among other things, the database relies on two file formats and data compression and is written in the C++ programming language. [2]

LevelDB is used for data persistence by the Chromium web browser, among others, and thus also by all web browsers based on it, such as Chrome, Microsoft Edge, Vivaldi, Brave, Opera, and Iridium.

The open-source project Chromium[2] was launched by Google in 2008 and not only forms the basis for many web browsers, but is also implemented, for example, by frameworks such as Electron[3] to enable operating system-independent desktop applications based on web technologies. Such applications include Visual Studio Code, Microsoft Teams[4], Slack, and WhatsApp Desktop.

Furthermore, common operating systems have now also integrated so-called WebViews, thereby enabling native applications to integrate web content via corresponding system interfaces. For example, the WebViews of Windows 10/11 and Android are also based on Chromium, which enables applications to persist data via LevelDB.

When using Chromium or Chromium-based web browsers or the respective WebViews of the operating systems, LevelDB is usually accessed indirectly via web interfaces (APIs[5]), such as the Web Storage API[4] (Session Storage/Local Storage) or the IndexedDB API[5]. In comparison, the Firefox web browser also uses the same web interfaces, but in this case stores the underlying data in SQLite databases rather than LevelDB.

However, LevelDB can also be embedded directly and natively in applications via the official program library, so that the database can be used without integrating Chromium. In addition to direct implementation in C++, frameworks such as Tauri, which is used in this work, enable access to the WebView of the respective operating system as well as the native implementation of libraries and thus also the use of the provided LevelDB library.

[1] Structured Query Language

[2] `https://www.chromium.org`

[3] `https://www.electronjs.org`

[4] from version 2.0 onwards, WebView2 is used instead of Electron, which is based on the Chromium-based Edge and continues to use LevelDB for data persistence [3, pp. 87 sqq.]

[5] Application Programming Interfaces

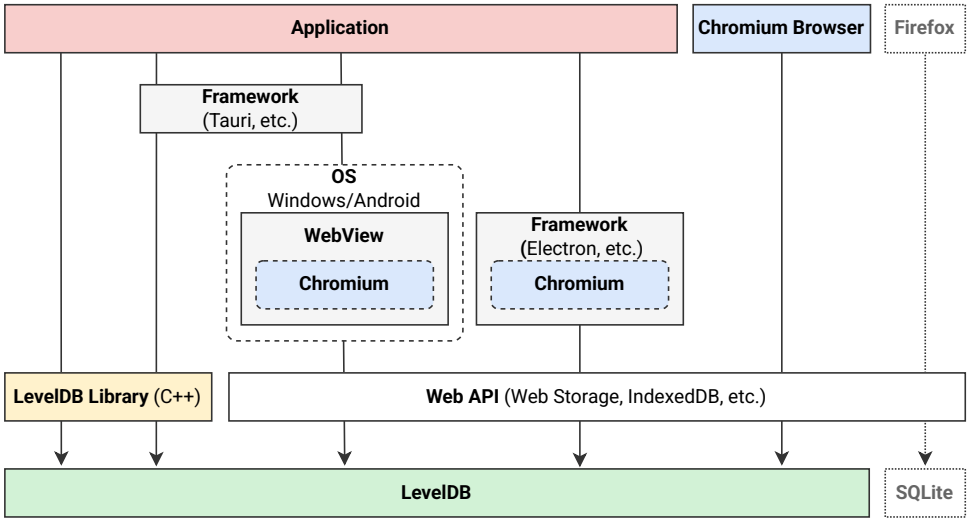



Figure 1.1: LevelDB scope of application

## 1.2 Problem Definition

In contrast to the SQLite database, which has a similar scope of application, LevelDB has been little researched in the field of digital forensics, despite its widespread use. For example, Professor Pawlaszczyk from Mittweida University of Applied Sciences describes in detail the structure and design of SQLite in the book *Mobile Forensics – The File Format Handbook* [6, pp. 129 sqq.] and also publishes a forensic open-source application for this database called FQLite[6].

On the subject of LevelDB, however, Alexander Bilz notes in his 2021 master's thesis *Digital Forensic Acquisition and Analysis of Artefacts Generated by Microsoft Teams* that many technical papers only deal with the database superficially and thus, for example, completely ignore compressed or deleted data.[3, pp. 27 sqq.] In his 2024 article *Google Chrome Platform Notification Analysis*, experienced digital forensics expert and instructor Chad Tilbury also writes about LevelDB that he cannot remember a time in his career when there has been such a significant gap in forensic capabilities for such an important application.[7]

No free standalone parsers for LevelDB that comply with forensic standards and disclose their source code could be found. Well-known and commonly used parsers, such as LevelDBDumper[7], are based on the official LevelDB library. However, this library does not output entries marked as deleted or other database information that may be relevant for digital forensics and goes beyond the mere entry data.

[6] `https://github.com/pawlaszczyk/fqlite`
[7] `https://github.com/mdawsonuk/LevelDBDumper`

On the commercial side, the only program available is LevelDB Recon[8] from the US company Arsenal Recon, which advertises its ability to extract deleted entries and other internally recorded information from the database. The commercial application RabbitHole[9] from the British company CCL Solutions Group is also said to be able to handle LevelDB data, among other things.

## 1.3 Objectives

In order to fill the gap for a free forensic standalone parser for LevelDB, this thesis aims to develop such an application and supplement it with a graphical user interface based on it.

In order to capture all existing entries, the extraction should not be performed via the official LevelDB library, but should be derived directly from the underlying data structure of the database files. Deleted and corrupt entries should be output as well, if possible, and highlighted as such.

Before implementation, a detailed analysis of the database structure and the data structures of the various and, in some cases, binary database files is required.

Finally, the parser should be subjected to various functional and performance tests as part of a validation and application example with real data.

The source code and the application should be documented on the GitHub platform[10] and published freely under the MIT license[11]. The permissive licensing allows free use and distribution of the software for private, scientific, and commercial purposes, as well as any modifications to the source code by third parties for their own purposes.

## 1.4 State of Research

The source code of LevelDB is freely available for viewing and analysis.[2] The official documentation[12] and implementation description of the database is very brief and streamlined to the essentials, so that many details about its functionality, structure, and design must be obtained through code analysis.

Digital forensics expert Alex Caithness has done a lot of preliminary work, particularly in describing the database structure and developing a parser script, as part of

[8] `https://arsenalrecon.com/additional-products`
[9] `https://www.cclsolutionsgroup.com/forensic-products/rabbithole`
[10] `https://github.com`
[11] `https://opensource.org/license/mit`
[12] `https://github.com/google/leveldb/tree/main/doc`

his work for the British IT security company CCL Solutions Group. Among other things, he described a large part of the database structure in his article *Hang on! That's not SQLite! Chrome, Electron and LevelDB*[8] dated September 23, 2020.

In addition, Alex Caithness developed the Chromium Reader application and library package[9], which includes the script `dump_leveldb.py`[13], which can extract entries marked as deleted from LevelDB in addition to other database-internal information.

Building on the work of Alex Caithness, Alexander Bilz developed a parser for the Microsoft Teams application as part of his master's thesis *Digital Forensic Acquisition and Analysis of Artefacts Generated by Microsoft Teams*[3][14] at the University of Abertay Dundee in Scotland in 2021, which he also makes available as a plugin for the forensic application Autopsy.[15]

## 1.5 Methodology

In addition to studying the preliminary work already done by Alex Caithness and Alexander Bilz, the structure, functionality, and design of LevelDB will be explored and described by analyzing the freely available source code.

The freely available development environment Qt Creator[16] will be used for the code and runtime analysis of the C++ code.

The knowledge gained in this way will be used to further refine the requirements for the planned parser application in order to create a basis for a design concept. The application should be able to extract and display all information stored in the database beyond the actual key-value entries, such as sequence numbers or existing entry data that is marked as deleted.

Based on this concept, the LevelDB parser will then be implemented to process the database contents according to the defined specifications.

The latest version 1.86.0 of the Rust programming language[17] will be used to implement the application. This modern programming language is considered to be high-performance and secure and is being promoted by authorities and companies as a possible successor to the aging C/C++ programming languages.[18] [19]

[13] `https://github.com/cclgroupltd/ccl_chromium_reader/blob/master/tools_and_utilities/dump_leveldb.py`

[14] `https://www.alexbilz.com/post/2021-09-09-forensic-artifacts-microsoft-teams`

[15] `https://github.com/lxndrblz/forensicsim`

[16] `https://www.qt.io/product/development-tools`

[17] `https://www.rust-lang.org`

[18] `https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues`

[19] `https://www.theregister.com/2022/09/20/rust_microsoft_c`

The application should be operated and displayed via a graphical user interface in addition to the command line. The cross-platform framework Tauri[20] is used for this purpose. This is based on Rust and uses web technologies, among other things, to display the user interface by accessing the native WebView of the respective operating system.

The data records extracted by the application should be displayed in tabular form. The respective data columns should be individually searchable to enable a clear and targeted content search. The open-source library AG Grid Community[21] is used for this purpose.

As part of the validation process, the completed application is to be tested on two different systems using various functional tests and, if necessary, adapted to ensure that the specified requirements are met.

Finally, the developed parser application is to be tested and presented using a real-world application example. For this purpose, the LevelDB database of the Microsoft Teams application is to be extracted and examined.

The main work and examinations are carried out under Fedora Linux 42 (Kernel: `6.14.3-300.fc42.x86_64`).

[20] `https://tauri.app`
[21] `https://www.ag-grid.com`

# Chapter 2

# Structure and Layout of LevelDB

LevelDB uses multiple files, each of which performs a specific task. A typical database directory consists of the following files:

```
leveldb
 ├── 000005.ldb      (binary)
 ├── 000006.log      (binary)
 ├── CURRENT         (text)
 ├── LOCK            (empty)
 ├── LOG             (text)
 ├── LOG.old         (text)
 └── MANIFEST-000004 (binary)
```

The database entries are stored in the binary files `.log` and `.ldb`. The data is first written to the `.log` file, which acts as a Write-Ahead Log (WAL) to ensure data security and consistency. Finally, the entry data is stored in files with the extension `.ldb`, which can occur multiple times and are not rewritable. The binary file `MANIFEST` contains metadata about the current state of the database. Since the file can occur multiple times, the text file `CURRENT` refers to the current manifest file. The file names `.log`, `.ldb`, and `MANIFEST` contain an incremental counter with leading zeros, which is incremented sequentially by one when the files are regenerated.[10]

The text file `LOG` or `LOG.old` stores the current or previous program outputs of the LevelDB process. When the database is opened, the LevelDB process locks the empty file `LOCK` so that no other LevelDB process can access the database.[10]

For further analysis, the source code of LevelDB was downloaded:

```
$ git clone --recurse-submodules https://github.com/google/leveldb.git

$ git rev-parse HEAD
ac691084fdc5546421a55b25e7653d450e5a25fb

$ git describe --tags
1.23-86-gac69108
```

The source code was then compiled as follows:

```
$ mkdir -p build && cd build
$ cmake -DCMAKE_BUILD_TYPE=Release .. && cmake --build .
```

In addition to the static library `libleveldb.a`, the build process also generates the command-line utility `leveldbutil`, which enables inspection of the binary database files.

## 2.1 Runtime Behavior

The following sections take a closer look at the runtime behavior of LevelDB with the help of a debugger in conjunction with static code analysis. For this purpose, an empty C++ project was created in Qt Creator with the integration of the LevelDB library.

The runtime behavior will be described in the following sections using typical database CRUD (Create, Read, Update, and Delete) operations. First, however, we will discuss the initial program behavior when creating or opening a LevelDB database.

### 2.1.1 Opening the Database

The database was created, opened, and closed again using the following code:

```
#include "leveldb/db.h"
int main() {
    leveldb::DB* db;
    leveldb::Options options;
    options.create_if_missing = true;
    leveldb::DB::Open(options, "./testdb", &db); // open database
    delete db; // close database
    return 0;
}
```

When executed, a directory with the specified database name (here: `testdb`) is created, if it does not already exist.

In the initial phase, the system searches for the file `LOG`. If this file does not exist, it is created empty. Otherwise, it is first renamed to `LOG.old` before being recreated. If a file with this name already exists, it is deleted first. This means that the LevelDB directory will only contain the logs from the previous execution and no older logs.

This is followed by the creation of the file `LOCK`, which is then locked exclusively by the process so that no other processes can access the database.

The creation of the database is noted in the text file `LOG` in the following form:

```
YYYY/MM/DD-HH:MM:SS.ffffff <Thread-ID> Creating DB ./testdb since it was missing.
```

This is followed by the creation of the file `MANIFEST-000001`. The contents of the binary file were read out with `leveldbutil` as follows:

```
$ leveldbutil dump MANIFEST-000001

--- offset 0; VersionEdit {
  Comparator: leveldb.BytewiseComparator
  LogNumber: 0
  NextFile: 2
  LastSeq: 0
}
```

The file contains VersionEdit lists that describe the current state of the database. By default, LevelDB uses the BytewiseComparator, which compares and sorts keys based on their byte order (lexicographical order). Other comparators are not available, but they can be implemented to change the key sorting logic.

The `LogNumber` entry records the current counter of the `.log` file. Since no `.log` file currently exists, a 0 is set here. The `NextFile` entry defines the next counter for the next manifest file. `LastSeq` records the last sequence number of a database entry used. Since no database entries have been written yet, the value is also set to 0 here.

This is followed by the creation of the file `CURRENT`[1], which contains the following text entry and thus points to the current manifest file:

```
MANIFEST-000001
```

[1] temporarily created as `000001.dbtmp` (corresponds to manifest number)

This is followed by the creation of the empty write-ahead log file `000003.log` and an empty MemTable, which is created in the main memory and is intended to hold the database entries.

The MemTable[2] is implemented as a skip list[11], which is basically a sorted linked list, but with additional levels in which individual key references are arranged randomly. This allows individual entries in the basis levels to be skipped during a search, enabling faster search operations.

Finally, a new `MANIFEST-000002` file is written with the following content:

```
$ leveldbutil dump MANIFEST-000002

--- offset 0; VersionEdit {
  Comparator: leveldb.BytewiseComparator
}
--- offset 35; VersionEdit {
  LogNumber: 3
  PrevLogNumber: 0
  NextFile: 4
  LastSeq: 0
}
```

The entry in the `CURRENT` file is updated to the current manifest file and the `MANIFEST-000001` file is deleted, which is noted in the log as follows:

```
Delete type=3 #1
```

During the second run, the file counter increases to `MANIFEST-000004` and `000005.log`, respectively. The `CURRENT` file points to the current manifest file, which refers to the current `.log` file (`LogNumber: 5`) and sets the next counter (`NextFile: 6`).

The file `LOG` contains the following entries:

```
Recovering log #3
Delete type=0 #3
Delete type=3 #2
```

Accordingly, the entries from the file `000003.log` are recovered first. In this case, the entries (here: none) from the old `.log` file would be written to a `.ldb` file via the MemTable before the old `.log` file (type=0) and the old manifest file (type=3) are deleted.

[2] `https://github.com/google/leveldb/blob/main/db/memtable.h`

Overall, the following initialization behavior was observed:

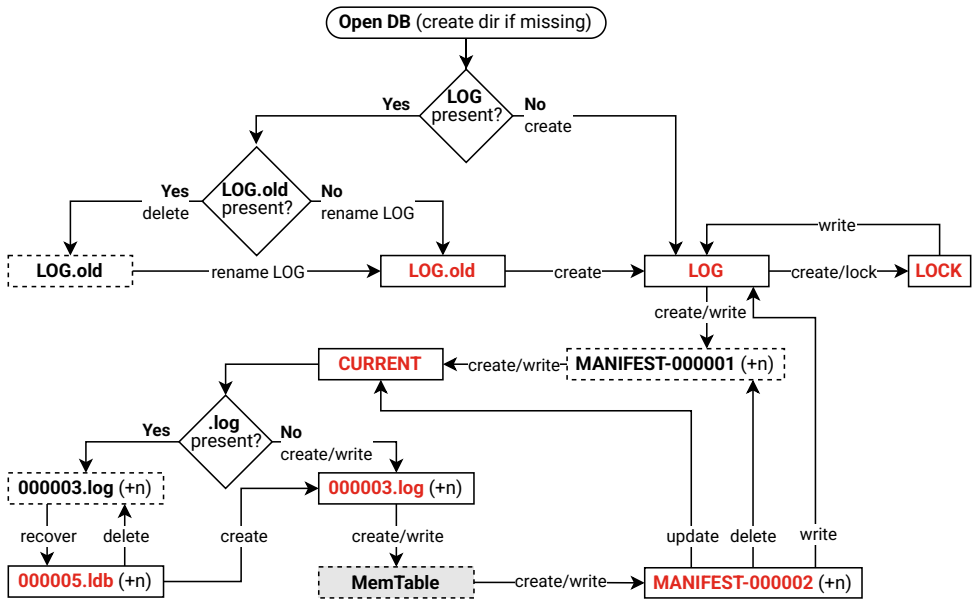



Figure 2.1: LevelDB initialization behavior

### 2.1.2 Creating an Entry

Before creating an entry, the database must be opened. After the entry has been created, the database is closed again, as shown in the program code in the previous section.

In the first pass, the first entry is created using the following code:

```
db->Put(leveldb::WriteOptions(), "Mozart", "Eine kleine Nachtmusik");
```

Here, the entry is first written to the WAL file `000003.log`, which was created during the initialization:

```
$ leveldbutil dump 000003.log

--- offset 0; sequence 1
  put 'Mozart' 'Eine kleine Nachtmusik'
```

The entry is then created in the main memory in the MemTable. No further relevant changes could be observed.

The second entry is then created as follows:

```
db->Put(leveldb::WriteOptions(), "Bach", "Air");
```

When the database is opened, the file `000003.log` created in the first run is first read in and its entries are stored in the file `000005.ldb` as part of the recovery process. The `.log` file is then deleted.

```
$ leveldbutil dump 000005.ldb

'Mozart' @ 1 : val => 'Eine kleine Nachtmusik'
```

The number after the @-sign corresponds to the sequence number of the entry.

The following entries are made in the log:

```
Recovering log #3
Level-0 table #5: started
Level-0 table #5: 140 bytes OK
```

The entries listed describe the creation of the non-rewritable file with sorted table `000005.ldb` with a total size of 140 bytes from the file `000003.log`.

The `.ldb` files are divided into **Levels** (here: Level-0), which is also the basis for the naming of LevelDB. The higher the level, the older the entries it contains. Level-0 files are the most recent tables, which still contain the complete data sets, including duplicate and deleted entries.

As soon as there are more than four Level-0 files, they are scheduled for a compaction process and written to Level-1 files. Duplicate and deleted entries are removed during this process.

Further $Level + n$ files are created as soon as the threshold value of $10^L$ for the combined file size in Level $L$ is exceeded. That is, 10 MB at *Level-1*, 100 MB at *Level-2*, and so on.[10]

Once the MemTable buffer exceeds 4 MiB, the MemTable entries are also written to a Level-0 file, after which a new MemTable and a new `.log` file are created.[3]

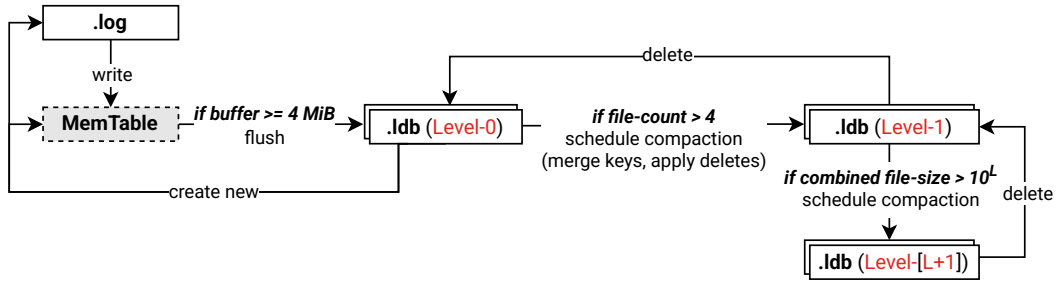



Figure 2.2: Life cycle of .ldb files

[3] `https://github.com/google/leveldb/blob/main/db/db_impl.cc#L455`

The recovery process is followed by the writing process of the current entry to the file `000006.log` and then to the MemTable:

```
$ leveldbutil dump 000006.log

--- offset 0; sequence 2
  put 'Bach' 'Air'
```

The manifest file is now structured as follows:

```
$ leveldbutil dump MANIFEST-000004

--- offset 0; VersionEdit {
  Comparator: leveldb.BytewiseComparator
}
--- offset 35; VersionEdit {
  LogNumber: 6
  PrevLogNumber: 0
  NextFile: 7
  LastSeq: 1
  AddFile: 0 5 140 'Mozart' @ 1 : 1 .. 'Mozart' @ 1 : 1
}
```

The `AddFile` entry shows that a Level-0 table (0) was written to the file `000005.ldb` (5) with a total size of 140 bytes. Since only one entry was saved, only the key `Mozart` is listed in the start (..) end key range, which has the sequence number 1 (@ 1) and is a regular, undeleted entry (: 1[4]).[5]

### 2.1.3 Reading an Entry

The key `Mozart` was read using the following code:

```
std::string value;
db->Get(leveldb::ReadOptions(), "Mozart", &value);
std::cout << "Output: " << value << std::endl;
```

The database output is written to the variable `value`, and the value for the queried key is then displayed:

```
Output: Eine kleine Nachtmusik
```

Before reading the entry, the entry `Bach` set in the previous section is first written from the `.log` file to the new table file `000008.ldb`:

[4] for deleted entries, a 0 is set instead

[5] `https://github.com/google/leveldb/blob/main/db/dbformat.h`

```
$ leveldbutil dump 000008.ldb

'Bach' @ 2 : val => 'Air'
```

The current `MANIFEST-000007` file now contains references to both table files:

```
$ leveldbutil dump MANIFEST-000007

--- offset 0; VersionEdit {
  Comparator: leveldb.BytewiseComparator
  AddFile: 0 5 140 'Mozart' @ 1 : 1 .. 'Mozart' @ 1 : 1
}
--- offset 70; VersionEdit {
  LogNumber: 9
  PrevLogNumber: 0
  NextFile: 10
  LastSeq: 2
  AddFile: 0 8 119 'Bach' @ 2 : 1 .. 'Bach' @ 2 : 1
}
```

When searching for a key, the MemTable is searched first. If the key is not found there, as in this example, the `.ldb` files are searched. In this case, the key `Mozart` was found in the table `000005.ldb`, whereupon the value is displayed.

### 2.1.4 Updating an Entry

The value of the key `Bach` has been updated as follows:

```
db->Put(leveldb::WriteOptions(), "Bach", "Das wohltemperierte Klavier");
```

When executed, the entry is made in the new log file `000011.log` with the sequence number incremented:

```
$ leveldbutil dump 000011.log

--- offset 0; sequence 3
  put 'Bach' 'Das wohltemperierte Klavier'
```

The newly created manifest file is now structured as follows:

```
$ leveldbutil dump MANIFEST-000010

--- offset 0; VersionEdit {
  Comparator: leveldb.BytewiseComparator
  AddFile: 0 8 119 'Bach' @ 2 : 1 .. 'Bach' @ 2 : 1
  AddFile: 0 5 140 'Mozart' @ 1 : 1 .. 'Mozart' @ 1 : 1
```

```
}
--- offset 100; VersionEdit {
  LogNumber: 11
  PrevLogNumber: 0
  NextFile: 12
  LastSeq: 2
}
```

Among other observations, it can be seen that the key `Bach` still refers to the old table with the previous sequence number, as the new entry is currently only contained in the `000011.log` file or the MemTable:

```
…
  AddFile: 0 8 119 'Bach' @ 2 : 1 .. 'Bach' @ 2 : 1
…
```

Only after another run will the new value be saved in the file `000013.ldb`:

```
$ leveldbutil dump 000013.ldb

'Bach' @ 3 : val => 'Das wohltemperierte Klavier'
```

The manifest file is now structured as follows and still contains the old key together with the assigned sequence number:

```
$ leveldbutil dump MANIFEST-000012

--- offset 0; VersionEdit {
  Comparator: leveldb.BytewiseComparator
  AddFile: 0 8 119 'Bach' @ 2 : 1 .. 'Bach' @ 2 : 1
  AddFile: 0 5 140 'Mozart' @ 1 : 1 .. 'Mozart' @ 1 : 1
}
--- offset 100; VersionEdit {
  LogNumber: 14
  PrevLogNumber: 0
  NextFile: 15
  LastSeq: 3
  AddFile: 0 13 143 'Bach' @ 3 : 1 .. 'Bach' @ 3 : 1
}
```

The old table file `000008.ldb` with the deleted value is still present and will only be removed during the next compaction process:

```
$ leveldbutil dump 000008.ldb

'Bach' @ 2 : val => 'Air'
```

### 2.1.5 Deleting an Entry

The key `Bach` was deleted with the following code after opening the database, followed by closing the database:

```
db->Delete(leveldb::WriteOptions(), "Bach");
```

The corresponding log file was then created as follows:

```
$ leveldbutil dump 000016.log

--- offset 0; sequence 4
  del 'Bach'
```

The updated manifest file is structured as follows:

```
$ leveldbutil dump MANIFEST-000015

--- offset 0; VersionEdit {
  Comparator: leveldb.BytewiseComparator
  AddFile: 0 13 143 'Bach' @ 3 : 1 .. 'Bach' @ 3 : 1
  AddFile: 0 8 119 'Bach' @ 2 : 1 .. 'Bach' @ 2 : 1
  AddFile: 0 5 140 'Mozart' @ 1 : 1 .. 'Mozart' @ 1 : 1
}

--- offset 131; VersionEdit {
  LogNumber: 16
  PrevLogNumber: 0
  NextFile: 17
  LastSeq: 3
}
```

The table file `000013.ldb`, which contains the deleted key and value, still exists:

```
$ leveldbutil dump 000013.ldb

'Bach' @ 3 : val => 'Das wohltemperierte Klavier'
```

During the next pass, the deletion of the key is stored in the created table file `000018.ldb`. The value of the key is not recorded, but an empty deletion marker is set, specifying the key name:

```
$ leveldbutil dump 000018.ldb

'Bach' @ 4 : del => ''
```

The current sequence number of the key and the deletion (value: 0) are recorded in the updated manifest file as follows:

```
$ leveldbutil dump MANIFEST-000017

--- offset 0; VersionEdit {
  Comparator: leveldb.BytewiseComparator
  AddFile: 0 13 143 'Bach' @ 3 : 1 .. 'Bach' @ 3 : 1
  AddFile: 0 8 119 'Bach' @ 2 : 1 .. 'Bach' @ 2 : 1
  AddFile: 0 5 140 'Mozart' @ 1 : 1 .. 'Mozart' @ 1 : 1
}

--- offset 131; VersionEdit {
  LogNumber: 19
  PrevLogNumber: 0
  NextFile: 20
  LastSeq: 4
  AddFile: 0 18 116 'Bach' @ 4 : 0 .. 'Bach' @ 4 : 0
}
```

Even after the last run, all table files were still available, meaning that both the changed and deleted values could be completely restored until a compaction process (see section 2.1.2) and traced chronologically using the sequence number.

The history of key changes or key deletions could still be traced in the respective manifest file using the sequence number and the subsequent value.

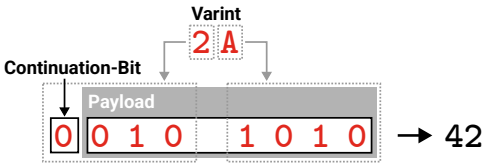
## 2.2 Structure and Layout of the Binary Files

The following section takes a closer look at the structure of the binary files used by LevelDB. These binary files are specifically the WAL files with the file extension `.log`, the sorted string table files with the file extension `.ldb` (`.sst`) and manifest files that record the current database status. Varint encoding is used in the binary files to save space when storing integers. This will be discussed in the next section before the actual binary files of the database are examined in detail.

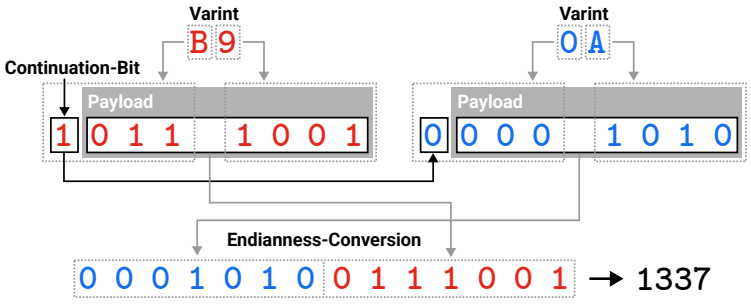
### 2.2.1 Variable-Width Integer (Varint)

Variable-width integers (Varints) comprise a technique for compact storage of integer values. Here, the values are encoded in a variable number of bytes. The smaller the value, the fewer bytes are required. 32-bit values can consist of one to five bytes, while 64-bit values can comprise up to ten bytes. Each byte contains seven data bits and one continuation bit (MSB[6]), which indicates whether more bytes follow (1) or not (0).[12]

For example, the number 42 (`0x2A`) is encoded in one byte as follows, corresponding to a regular integer encoding:




A larger number, on the other hand, requires several bytes. The bytes are traversed until the continuation bit corresponds to the value 0. The data bits are then assembled, which, after an endianness conversion, result in the actual value:




[6]Most Significant Bit

In varint encoding, LevelDB uses only unsigned, positive integers (uint32/uint64):

https://github.com/google/leveldb/blob/main/util/coding.h#L48-L49

```
48 char* EncodeVarint32(char* dst, uint32_t value);
49 char* EncodeVarint64(char* dst, uint64_t value);
```

If, on the other hand, negative numbers are to be encoded as varints—as is possible, for example, by the Protobuf implementation (see section 3.1)—the problem arises that negative numbers set the most significant bit in two's complement. For example, in the case of a small negative 32-bit number such as `-1`, all bits would be set, meaning that the maximum number of bytes would be required for such numbers. A `-1` would therefore require all five bytes as a varint instead of just one byte.

To circumvent this problem, Protobuf, for example, uses ZigZag encoding for signed numbers (sint32/sint64), in which positive numbers are doubled (the number becomes even) and negative numbers are doubled and reduced by one (the number becomes odd), with the sign being omitted so that only positive numbers need to be transferred with as few bytes as possible and a reconversion can be performed at the end according to the data type.[12]

### 2.2.2 .log

The `.log` file is a write-ahead log whose file name consists of an incremental counter that is incremented globally for the `.log`, `.ldb`, and `MANIFEST` files.

As stated in section 2.1.1, the database data is first written to the `.log` file and then stored in the MemTable in the main memory. This is to ensure the security and consistency of the data so that it can be restored from the `.log` file in the event of a failure, for example.

After closing and reopening the database, the data from the `.log` file of the previous session is written to a `.ldb` file and a new `.log` file is created for the current session (see section 2.1.1). This also happens as soon as the file size exceeds 4 MiB (see section 2.1.2).

Writing to the `.log` file is done sequentially without changing the previously recorded data. This means that all data changes, such as writing, updating, and deleting entries in the file, are continuously documented and can be traced chronologically.

The `.log` file is divided into 32 KiB blocks:

https://github.com/google/leveldb/blob/main/db/log_format.h#L27

```
27 static const int kBlockSize = 32768;
```

If the block size is exceeded by an entry, the entry is split across multiple blocks and the respective block type is set accordingly:

https://github.com/google/leveldb/blob/main/db/log_format.h#L14-L24

```
14 enum RecordType {
15   // Zero is reserved for preallocated files
16   kZeroType = 0,
17
18   kFullType = 1,
19
20   // For fragments
21   kFirstType = 2,
22   kMiddleType = 3,
23   kLastType = 4
24 };
```

The individual entries are preceded by a 7-byte header consisting of the masked checksum (CRC-32C) of the entry, block length, and the block type described above:

https://github.com/google/leveldb/blob/main/db/log_format.h#L29-L30

```
29 // Header is checksum (4 bytes), length (2 bytes), type (1 byte).
30 static const int kHeaderSize = 4 + 2 + 1;
```

The checksum is masked or unmasked as follows to avoid possible collisions:

https://github.com/google/leveldb/blob/main/util/crc32c.h#L22-L38

```
22 static const uint32_t kMaskDelta = 0xa282ead8ul;
23
24 // Return a masked representation of crc.
25 //
26 // Motivation: it is problematic to compute the CRC of a string that
27 // contains embedded CRCs.  Therefore we recommend that CRCs stored
28 // somewhere (e.g., in files) should be masked before being stored.
29 inline uint32_t Mask(uint32_t crc) {
30   // Rotate right by 15 bits and add a constant.
31   return ((crc >> 15) | (crc << 17)) + kMaskDelta;
32 }
33
34 // Return the crc whose masked representation is masked_crc.
35 inline uint32_t Unmask(uint32_t masked_crc) {
36   uint32_t rot = masked_crc - kMaskDelta;
37   return ((rot >> 17) | (rot << 15));
38 }
```

If it is a complete entry (record type 1) or the first entry in a series (record type 2), it is followed by a 12-byte *WriteBatch header*, which consists of the sequence number and the entry counter:

https://github.com/google/leveldb/blob/main/db/write_batch.cc#L26-L27

```
26 // WriteBatch header has an 8-byte sequence number followed by a 4-byte count.
27 static const size_t kHeader = 12;
```

The WriteBatch header is followed by a byte containing the status of the entry, which is set to 0 for deleted entries and 1 for other entries:

https://github.com/google/leveldb/blob/main/db/dbformat.h#L54

```
54 enum ValueType { kTypeDeletion = 0x0, kTypeValue = 0x1 };
```

Finally, the actual key or value is set, whose byte length is stored as varint32:

https://github.com/google/leveldb/blob/main/util/coding.cc#L72-L75

```
72 void PutLengthPrefixedSlice(std::string* dst, const Slice& value) {
73   PutVarint32(dst, value.size());
74   dst->append(value.data(), value.size());
75 }
```

The following code is used to write, modify, or delete an entry:

https://github.com/google/leveldb/blob/main/db/write_batch.cc#L98-L109

```
98  void WriteBatch::Put(const Slice& key, const Slice& value) {
99    WriteBatchInternal::SetCount(this, WriteBatchInternal::Count(this) + 1);
100   rep_.push_back(static_cast<char>(kTypeValue));
101   PutLengthPrefixedSlice(&rep_, key);
102   PutLengthPrefixedSlice(&rep_, value);
103 }
104
105 void WriteBatch::Delete(const Slice& key) {
106   WriteBatchInternal::SetCount(this, WriteBatchInternal::Count(this) + 1);
107   rep_.push_back(static_cast<char>(kTypeDeletion));
108   PutLengthPrefixedSlice(&rep_, key);
109 }
```

A deleted entry is thus represented by the key name, which does not contain its value and whose status byte is set to 0.

Code analysis revealed the following structure in `.log` files:

| Offset | Length | Description |
|---|---|---|
| 0 | 4 | Checksum (masked CRC-32C) |
| 4 | 2 | Data size |
| 6 | 1 | Record type:<br>1 -> Full; 2 -> First; 3 -> Middle; 4 -> Last |
| 7 | 8 | Sequence number |
| 15 | 4 | Record count |
| 19 | 1 | Record state:<br>0 -> Deleted; 1 -> Live |
| 20 | 1-5 | Key size (varint32) |
| ... | ... | Key (blob) |
| ... | 1-5 | Value size (varint32) |
| ... | ... | Value (blob) |

Figure 2.3: Block structure of .log files

The entries generated in the `.log` file as part of the runtime analysis in section 2.1 can therefore be interpreted as follows:

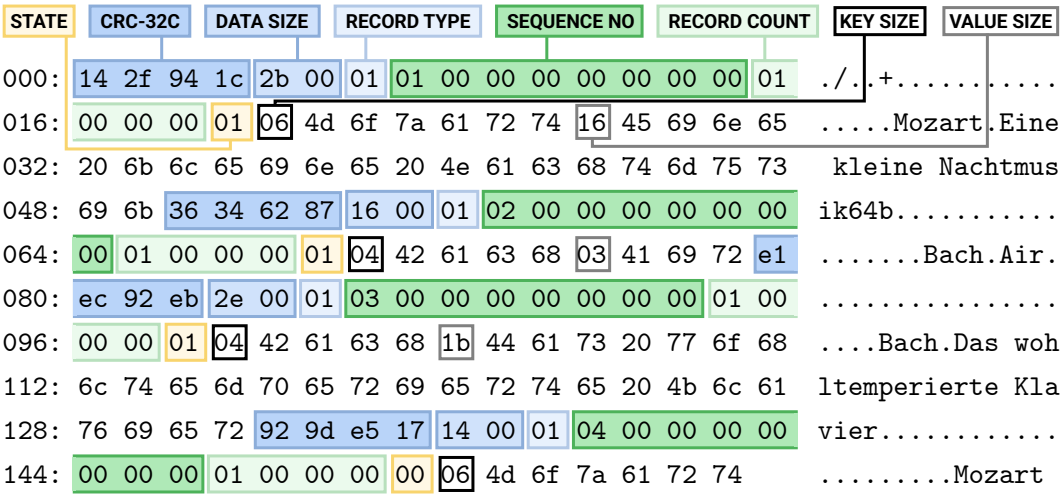



Figure 2.4: Sample data structure (.log) as per section 2.1

The sequence number and entry counter show that there are four individual entries in total. The entry status of the last entry indicates that it has been deleted (status 0). The value of the deleted key can be found in the first entry. Entries two and three have the same key, so the ascending sequence number shows that the value of the key in sequence number 2 has been changed to the value in sequence number 3.

### 2.2.3 .ldb/.sst

`.ldb` files are sorted, non-rewritable table files (Sorted String Table [SSTable]). LevelDB generally creates these with the file extension `.ldb`, but also accepts the file extension `.sst`:

https://github.com/google/leveldb/blob/main/db/filename.cc#L111-L113

```
111     } else if (suffix == Slice(".sst") || suffix == Slice(".ldb")) {
112       *type = kTableFile;
113     } …
```

The table file consists of several blocks. The **Data Blocks** at the beginning of the file contain the actual database entries. The data blocks can be followed by an optional **Meta Block**. Currently, only the *Filter Meta Block* is implemented, which can contain filter structures for efficient searching. In addition, the implementation of a *Stats Meta Block* is planned, which will provide a statistical overview of the table file. This is followed by a **Metaindex Block**, which refers to the meta block, and an **Index Block**, which contains the references to the individual data blocks. The file is concluded by a **Footer**.[13]

For the references/pointers to the external blocks, the Footer, the Index Block, and the Meta Index Block use **Block Handles**, each of which contains the position and size of the respective blocks, which are encoded as Varin64:

https://github.com/google/leveldb/blob/main/table/format.cc#L16-L22

```
16  void BlockHandle::EncodeTo(std::string* dst) const {
17    // Sanity check that all fields have been set
18    assert(offset_ != ~static_cast<uint64_t>(0));
19    assert(size_ != ~static_cast<uint64_t>(0));
20    PutVarint64(dst, offset_);
21    PutVarint64(dst, size_);
22  }
```

Except for the footer, the respective blocks consist of the **block data**, the **compression type**, and a masked **checksum** (CRC-32C, see section: 2.2.2):

https://github.com/google/leveldb/blob/main/table/table_builder.cc

```
141 void TableBuilder::WriteBlock(BlockBuilder* block, BlockHandle* handle) {
142   // File format contains a sequence of blocks where each block has:
143   //    block_data: uint8[n]
144   //    type: uint8
145   //    crc: uint32
```

Except for the *Meta Block*, which has its own structure, the entries in the **block data** consist of *Shared Bytes*, *Unshared Bytes*, *Value Size*, the key, and the value:

`https://github.com/google/leveldb/blob/main/table/block_builder.cc#L16-L22`

```
16 // An entry for a particular key-value pair has the form:
17 //     shared_bytes: varint32
18 //     unshared_bytes: varint32
19 //     value_length: varint32
20 //     key_delta: char[unshared_bytes]
21 //     value: char[value_length]
22 // shared_bytes == 0 for restart points.
```

An eight-byte suffix in the form of the **internal key** is appended to the key, which contains the status (1 → live / 0 → deleted) and the sequence number of the entry:

`https://github.com/google/leveldb/blob/main/db/dbformat.cc#L21-L24`

```
101 void AppendInternalKey(std::string* result, const ParsedInternalKey& key) {
102   result->append(key.user_key.data(), key.user_key.size());
103   PutFixed64(result, PackSequenceAndType(key.sequence, key.type));
104 }
```

If keys begin with the same bytes, only the differing bytes in the current key are stored to minimize storage space requirements, and the initial bytes of the preceding key are referenced with the length entry under *Shared Bytes*.

Block data can consist of multiple entries, provided that the size does not exceed 4 KiB:

`https://github.com/google/leveldb/blob/main/include/leveldb/options.h#L101`

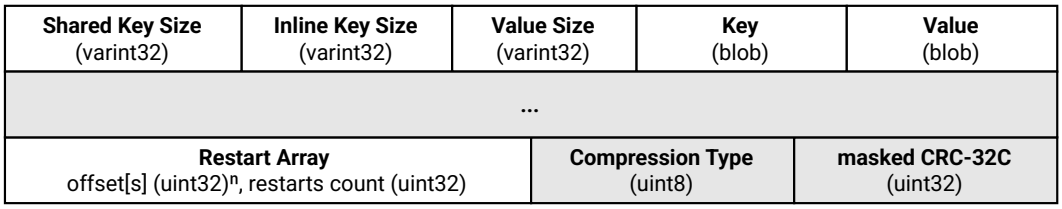
```
101   size_t block_size = 4 * 1024;
```

The block data is concluded with a trailer in the form of the **Restart Array**, which stores the positions of all full keys (shared bytes → 0) in the respective block to enable efficient searching:

`https://github.com/google/leveldb/blob/main/table/block_builder.cc#L24-L27`

```
24 // The trailer of the block has the form:
25 //     restarts: uint32[num_restarts]
26 //     num_restarts: uint32
27 // restarts[i] contains the offset within the block of the ith restart point.
```

The following block structure was therefore identified:




By default, LevelDB uses the *Snappy* library developed by Google to compress block data. Alternatively, *Zstd* can also be configured. Depending on the compression type used, the byte is set as follows:

`https://github.com/google/leveldb/blob/main/include/leveldb/options.h`

```
25 enum CompressionType {
26   // NOTE: do not change the values of existing entries, as these are
27   // part of the persistent format on disk.
28   kNoCompression = 0x0,
29   kSnappyCompression = 0x1,
30   kZstdCompression = 0x2,
31 };
```

Compression is used when the block data can be reduced by 12.5 % or more:

`https://github.com/google/leveldb/blob/main/table/table_builder.cc#L158-L170`

```
158     case kSnappyCompression: {
159       std::string* compressed = &r->compressed_output;
160       if (port::Snappy_Compress(raw.data(), raw.size(), compressed) &&
161           compressed->size() < raw.size() - (raw.size() / 8u)) {
162         block_contents = *compressed;
163       } else {
164         // Snappy not supported, or compressed less than 12.5%, so just
165         // store uncompressed form
166         block_contents = raw;
167         type = kNoCompression;
168       }
169       break;
170     }
```

During the search, the table file is scanned from bottom to top, starting with the **Footer**, which contains the block handles for the index block and the meta index block. The two block handles are concluded by the last eight bytes, which consist of the magic number `57 FB 80 8B 24 75 47 DB` (hex, little-endian) and are used to identify the table file:

https://github.com/google/leveldb/blob/main/table/format.h#L73-L76

```
73 // kTableMagicNumber was picked by running
74 //    echo http://code.google.com/p/leveldb/ | sha1sum
75 // and taking the leading 64 bits.
76 static const uint64_t kTableMagicNumber = 0xdb4775248b80fb57ull;
```

Above the footer is the **Index Block**, which refers to the database entries in the data blocks. It contains one entry per data block. The key consists of a separator that is greater than or equal to the entry key in the data block and is used for efficient searching. It is filled with the byte `0xFF`. The value consists of a block handle to the respective data block:

https://github.com/google/leveldb/blob/main/table/table_builder.cc

```
243   // Write index block
244   if (ok()) {
245     if (r->pending_index_entry) {
246       r->options.comparator->FindShortSuccessor(&r->last_key);
247       std::string handle_encoding;
248       r->pending_handle.EncodeTo(&handle_encoding);
249       r->index_block.Add(r->last_key, Slice(handle_encoding));
250       r->pending_index_entry = false;
251     }
252     WriteBlock(&r->index_block, &index_block_handle);
253   }
```

Above the index block is the **Meta Index Block**, which refers to optional meta blocks. The key consists of the name of the meta block and the value consists of the block handle for the respective meta block:

https://github.com/google/leveldb/blob/main/table/table_builder.cc

```
227   // Write metaindex block
228   if (ok()) {
229     BlockBuilder meta_index_block(&r->options);
230     if (r->filter_block != nullptr) {
231       // Add mapping from "filter.Name" to location of filter data
232       std::string key = "filter.";
233       key.append(r->options.filter_policy->Name());
234       std::string handle_encoding;
235       filter_block_handle.EncodeTo(&handle_encoding);
236       meta_index_block.Add(key, handle_encoding);
237     }
238
239     // TODO(postrelease): Add stats and other meta blocks
240     WriteBlock(&meta_index_block, &metaindex_block_handle);
241   }
```

A filter block can currently be activated in LevelDB as a meta block, which provides a **Bloom Filter**. This checks whether a table entry may exist or definitely does not exist. If the result is negative, the table does not need to be searched, which speeds up the search. If the filter has been activated, the following key naming occurs in the meta index block:

`https://github.com/google/leveldb/blob/main/util/bloom.cc#L26`

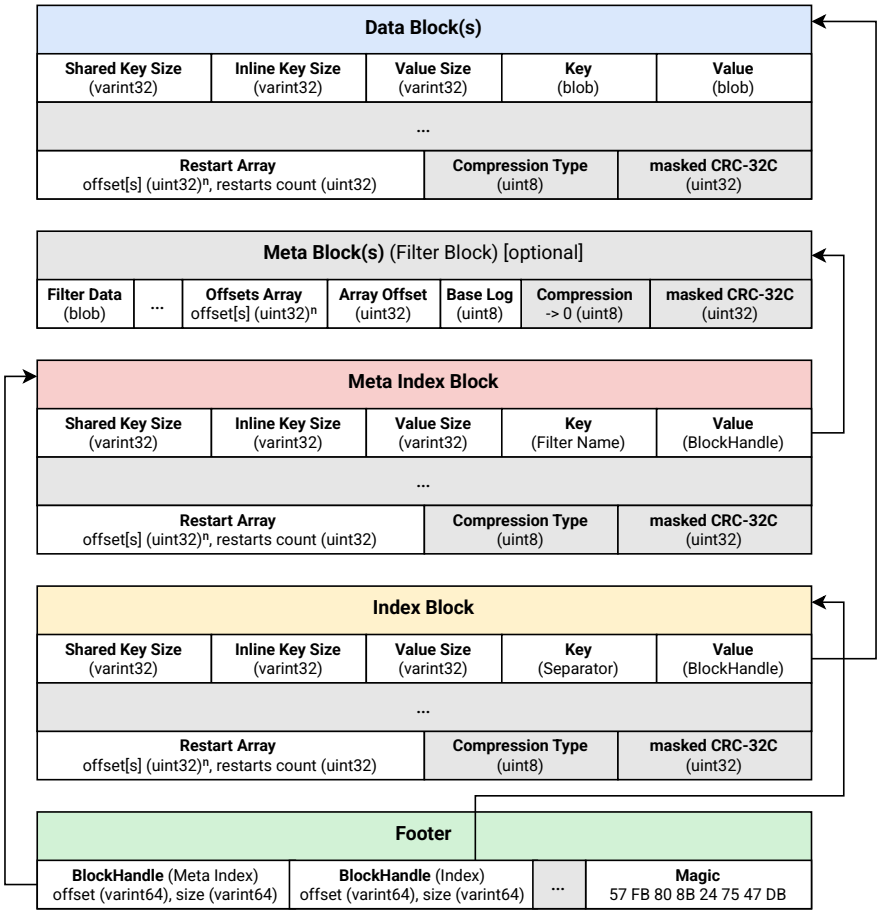
```
26  const char* Name() const override { return "leveldb.BuiltinBloomFilter2"; }
```

When the filter is activated, a meta block without compression and with its own structure is created above the meta index block:

`https://github.com/google/leveldb/blob/main/table/table_builder.cc`

```
221  // Write filter block
222  if (ok() && r->filter_block != nullptr) {
223    WriteRawBlock(r->filter_block->Finish(), kNoCompression,
224                  &filter_block_handle);
225  }
```

Analysis of the code has revealed the following structure of `.ldb` files:



For illustration purposes, a `.ldb` file was created with the following code:

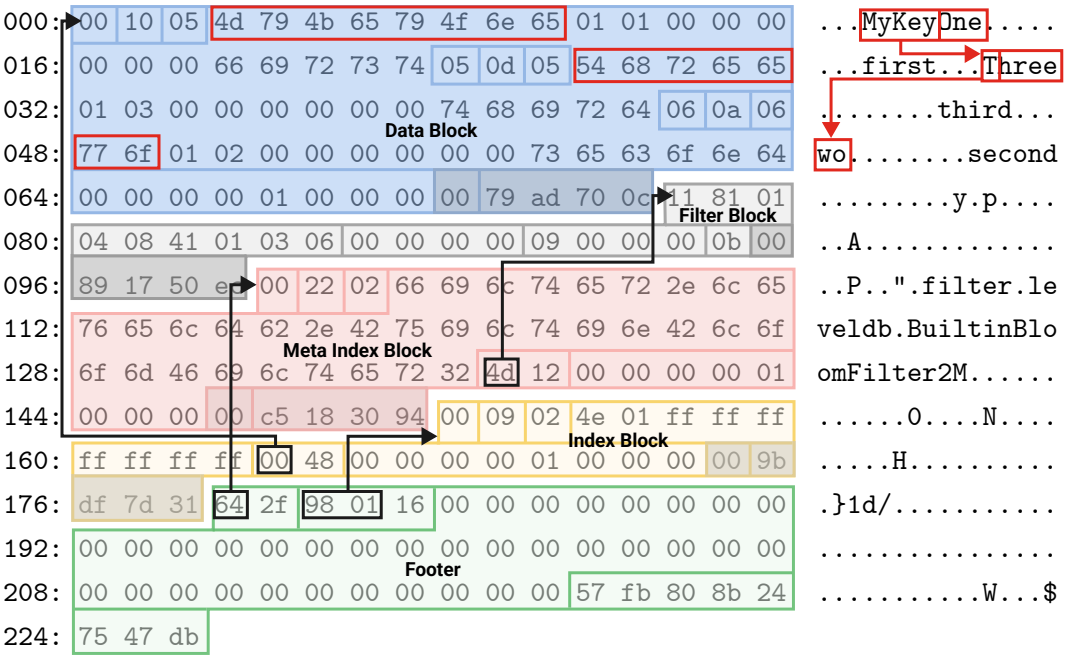
```
#include "leveldb/db.h"
#include "leveldb/filter_policy.h"
int main() {
    leveldb::DB *db;
    leveldb::Options options; options.create_if_missing = true;
    options.filter_policy = leveldb::NewBloomFilterPolicy(10); // bloom filter

    leveldb::DB::Open(options, "./testdb", &db);

    db->Put(leveldb::WriteOptions(), "MyKeyOne", "first");
    db->Put(leveldb::WriteOptions(), "MyKeyTwo", "second");
    db->Put(leveldb::WriteOptions(), "MyKeyThree", "third");

    delete options.filter_policy; delete db;
    return 0;
}
```

The `.ldb` file generated in this way has the following structure:




The footer refers to the two index blocks using the block handles. The meta index block contains the identifier of the activated Bloom filter and refers to it, while the index block refers to the data block containing the three database entries. The keys were not sorted by input order, but by their byte sequence. Since the three keys begin with the same bytes, compression was performed and the corresponding number of shared bytes to the preceding key was indicated.

### 2.2.4 MANIFEST

The manifest file contains current database information about the present `.log` file and the `.ldb` files, as well as their levels and the respective key ranges recorded.

The same format is used for the structure as for `.log` files (see section 2.2.2). Accordingly, the individual entries are preceded by a 7-byte header consisting of the checksum, the entry size, and the status byte.

The header is followed by entries consisting of the *tags* listed below, which are encoded using the `VersionEdit::EncodeTo` method:

`https://github.com/google/leveldb/blob/main/db/version_edit.cc#L14-L24`

```
14 enum Tag {
15   kComparator = 1,
16   kLogNumber = 2,
17   kNextFileNumber = 3,
18   kLastSequence = 4,
19   kCompactPointer = 5,
20   kDeletedFile = 6,
21   kNewFile = 7,
22   // 8 was used for large value refs
23   kPrevLogNumber = 9
24 };
```

The **Comparator-Tag** (1) is created as follows and only supports the Bytewise Comparator by default (see section 2.1.1):

`https://github.com/google/leveldb/blob/main/db/version_edit.cc#L42-L46`

```
42 void VersionEdit::EncodeTo(std::string* dst) const {
43   if (has_comparator_) {
44     PutVarint32(dst, kComparator);
45     PutLengthPrefixedSlice(dst, comparator_);
46   }
```

**LogNumber-Tag** (2):

`https://github.com/google/leveldb/blob/main/db/version_edit.cc#L47-L50`

```
47   if (has_log_number_) {
48     PutVarint32(dst, kLogNumber);
49     PutVarint64(dst, log_number_);
50   }
```

**NextFileNumber-Tag** (3):

https://github.com/google/leveldb/blob/main/db/version_edit.cc

```
55  if (has_next_file_number_) {
56    PutVarint32(dst, kNextFileNumber);
57    PutVarint64(dst, next_file_number_);
58  }
```

**LastSequence-Tag** (4):

https://github.com/google/leveldb/blob/main/db/version_edit.cc

```
59  if (has_last_sequence_) {
60    PutVarint32(dst, kLastSequence);
61    PutVarint64(dst, last_sequence_);
62  }
```

**CompactPointer-Tag** (5):

https://github.com/google/leveldb/blob/main/db/version_edit.cc

```
64  for (size_t i = 0; i < compact_pointers_.size(); i++) {
65    PutVarint32(dst, kCompactPointer);
66    PutVarint32(dst, compact_pointers_[i].first);  // level
67    PutLengthPrefixedSlice(dst, compact_pointers_[i].second.Encode());
68  }
```

**DeletedFile-Tag** (6):

https://github.com/google/leveldb/blob/main/db/version_edit.cc

```
70  for (const auto& deleted_file_kvp : deleted_files_) {
71    PutVarint32(dst, kDeletedFile);
72    PutVarint32(dst, deleted_file_kvp.first);   // level
73    PutVarint64(dst, deleted_file_kvp.second);  // file number
74  }
```

**NewFile-Tag** (7):

https://github.com/google/leveldb/blob/main/db/version_edit.cc

```
76  for (size_t i = 0; i < new_files_.size(); i++) {
77    const FileMetaData& f = new_files_[i].second;
78    PutVarint32(dst, kNewFile);
79    PutVarint32(dst, new_files_[i].first);  // level
80    PutVarint64(dst, f.number);
81    PutVarint64(dst, f.file_size);
82    PutLengthPrefixedSlice(dst, f.smallest.Encode());
83    PutLengthPrefixedSlice(dst, f.largest.Encode());
84  }
```

**PrevLogNumber-Tag** (9):

https://github.com/google/leveldb/blob/main/db/version_edit.cc

```
51  if (has_prev_log_number_) {
52    PutVarint32(dst, kPrevLogNumber);
53    PutVarint64(dst, prev_log_number_);
54  }
```

The file `MANIFEST-000017`, which was generated during the runtime analysis in section 2.1.5 can therefore be interpreted as follows:

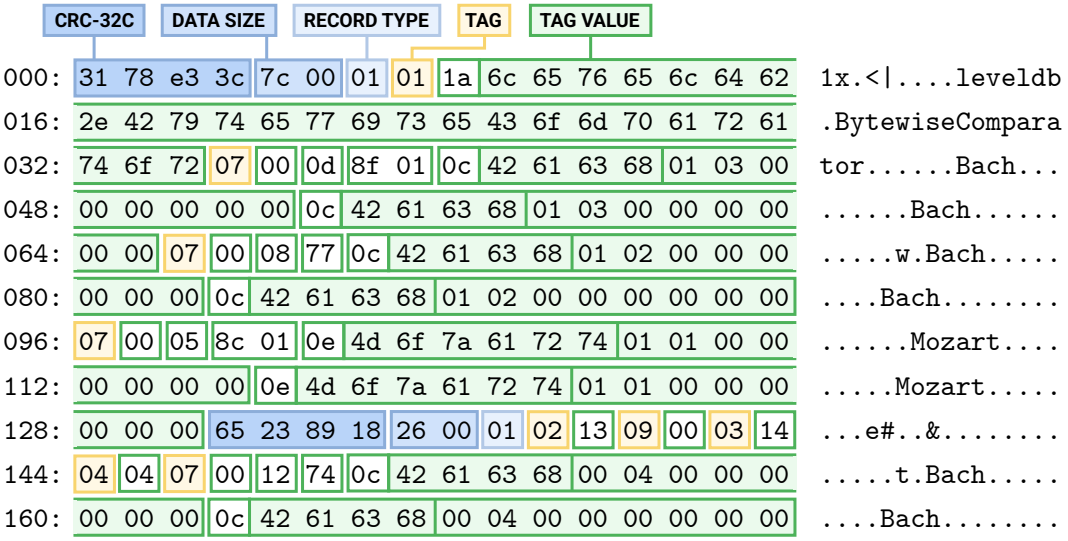



Figure 2.5: Structure of a manifest file

The file consists of two blocks. The first block contains the standard comparator tag (tag 1), entries for three table files (tag 7), each of which has level 0 as well as the key range, sequence number, and key status.

The second block contains another entry for a level 0 table file with a deleted key, as well as four metadata entries for the log number (tag 2), the numbering of the next table file (tag 3), the last sequence number used (tag 4), and the previous log number (tag 9).

# Chapter 3

# Excursus: LevelDB in Chromium-Based Applications

LevelDB is often used in Chromium-based applications, which is why it will be discussed in more detail in this excursus. As an instance of a web browser or an operating system WebView, or an application that implements such an instance, Chromium offers the possibility of data persistence in LevelDB via corresponding web interfaces (APIs).

It is up to the developer of such an application to decide whether and which web interfaces to use, provided that the technical specifications and limitations of the respective interfaces are considered. The Web Storage API is typically used for storing and retrieving simple key-value pairs, while the IndexedDB API is used for larger and more complex data structures.

For example, Alexander Bilz examined the Microsoft Teams application and concluded that it largely utilizes the IndexedDB API, but also utilizes the Web Storage API.[3, pp. 47 sqq.] The examination of the Android application Notion, on the other hand, revealed that it increasingly utilizes SQLite, but also accesses the Web Storage API.[1, pp. 11 sqq.]

In the following, the aforementioned interfaces will be examined in more detail using the Chromium web browser, which is representative of numerous applications based on it. In particular, the databases generated in each case will be examined in a forensic context. First, however, we will discuss the Protobuf encoding that is ubiquitous in Google applications and is also used by the web interfaces mentioned above.

## 3.1 Protocol Buffers (Protobuf)

Google's Protocol Buffers (Protobuf) serialization format is used—similar to JSON[1]—for storing and transferring key-value information. Unlike the human-readable JSON format, Protobuf is a binary format.[12]

Information about key names and value types is not transferred, but is stored in separate definition files in text form, which usually have the file extension `.proto`.[12]

These definitions can and are usually compiled for the respective application programming language using a separate Protobuf compiler[2] so that the data to be transferred can be interpreted natively by the application.[12]

A definition is introduced by the version number used (currently version 3) and then contains one or more named definition blocks, with the main block usually beginning with the keyword `message`.[12]

Definition blocks contain individual fields consisting of the field type, field name, and an ID (field number) assigned to the respective key:[12]

```
syntax = "<proto-version>";

message <block-name> {
    <field-type> <field-name> = <field-number>;
}
```

For example, to transfer information about a book, a definition in the form of the file `book.proto` could look like this:

```
syntax = "proto3";

message Book {
    string title  = 1;
    string author = 2;
    int32  year   = 3;
    float  price  = 4;
}
```

[1] JavaScript Object Notation
[2] `https://protobuf.dev/installation/`

In order to transfer the relevant information, it must first be encoded in Protobuf format. The Protobuf compiler can be used for this purpose, for example, by passing it the data to be transferred together with the definition file:

```
$ echo 'title: "Neuromancer" author: "Gibson" year: 1984 price: 19.99' | \
    protoc --encode=Book book.proto > data.bin
```

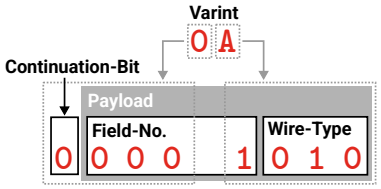
The command listed generates the binary file `data.bin` with the following content:

```
HEX                                                  | ASCII
-----------------------------------------------------------------------
0a 0b 4e 65 75 72 6f 6d 61 6e 63 65 72 12 06 47  | ..Neuromancer..G
69 62 73 6f 6e 18 c0 0f 25 85 eb 9f 41           | ibson...%...A
```

As can be seen, strings are stored as such and encoded in UTF-8. The other stored data is binary encoded.

The individual entries are preceded by an identifier that is encoded as a varint and contains the entry ID (field number) and the transmission type (wire type).

The sample data begins with the identifier `0x0a`. The first bit (continuation bit) is set to 0, meaning that no further bytes belong to this varint. The next four bits indicate the entry ID (here: 1). The last three bits represent the transfer type[3] (here: 2):



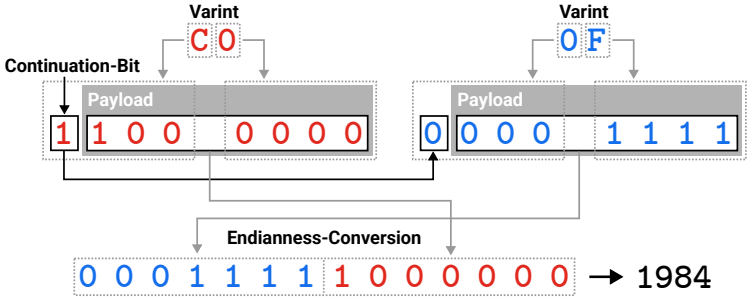

A total of six transfer types are defined, with type 0 (VARINT: integers) and type 2 (LEN [Length-Delimited Records]: including strings) being the most common. Accordingly, protobuf-encoded data often begins with the byte `0x0a` for strings, as shown here, or with the byte `0x08` for integers.

Accordingly, the first entry in the sample data is of type 2 (length-delimited record). The following byte indicates the length of the entry for this type (here: `0x0B` → 11).

The eleven bytes of the string of the first entry are followed by the second entry with the varint `0x12` → `0b00010010`, which is therefore also type 2. The next byte `0x06` indicates the length of the following string.

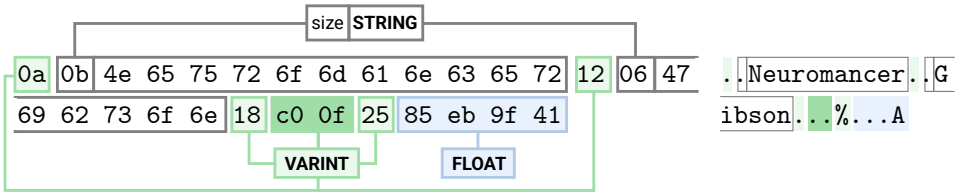
[3] `https://protobuf.dev/programming-guides/encoding/#structure`

The third entry is marked by the varint `0x18` → `0b00011000`, which is therefore transfer type 0 (varint). The continuation bit of the following byte `0xC0` → `0b11000000` is set to 1, so that the byte `0x0F` → `0b00001111` also represents a part of the data content:




The fourth and last entry is marked by the varint `0x25` → `0b00100101`. The transfer type is set to 5 (I32) here, which covers integers and floating point numbers that are encoded with a fixed four bytes (here: `0x85 0xEB 0x9F 0x41`). The definition file shows that this entry has been assigned the type *float*, which, after endianness conversion, corresponds to the rounded floating point number 19.99.

Manual analysis of the Protobuf data revealed the following encoding:

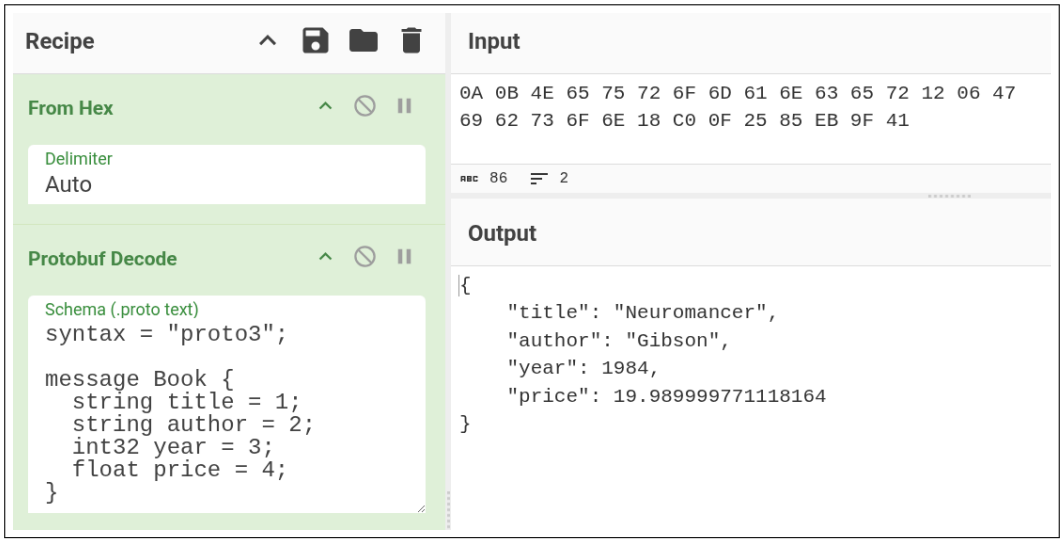



Of course, Protobuf data does not have to be decoded manually, but can be done using the Protobuf compiler, for example. If the definitions are available, the data can be decoded completely using the `--decode` option as follows:

```
$ protoc --decode=Book book.proto < data.bin

title: "Neuromancer"
author: "Gibson"
year: 1984
price: 19.99
```

CyberChef[4] also offers the option of decoding protobuf-encoded data:




If the definitions are not available, the data can be decoded using the Protobuf compiler and the `--decode_raw` option, whereby the floating point number is displayed in hex format, as the associated data type is unknown in this case:

```
$ protoc --decode_raw < data.bin

1: "Neuromancer"
2: "Gibson"
3: 1984
4: 0x419feb85
```

The Protobuf-Inspector[5] application by Alba Mendez offers a somewhat more comprehensive decoding option for protobuf-encoded data, including output of the assigned data types and possible interpretations of the values:

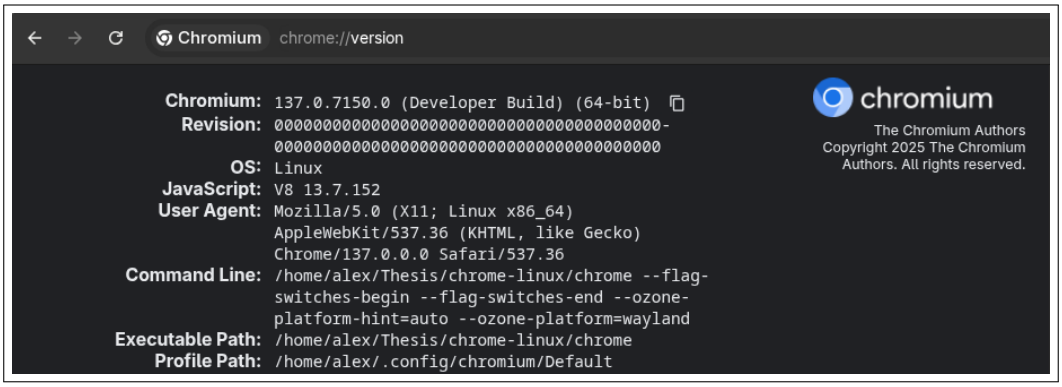
```
$ protobuf_inspector < data.bin

root:
    1 <chunk> = "Neuromancer"
    2 <chunk> = "Gibson"
    3 <varint> = 1984
    4 <32bit> = 0x419FEB85 / 1100999557 / 19.9900
```

[4] https://gchq.github.io/CyberChef
[5] https://github.com/mildsunrise/protobuf-inspector

## 3.2 Chromium Web Browser

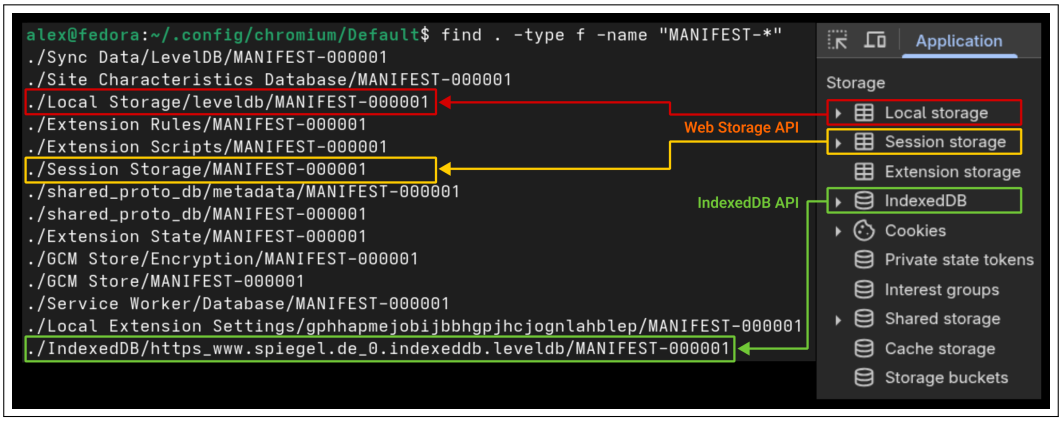
The latest version of Chromium was downloaded from the official source[6]. The web browser does not require separate installation/configuration and can be run directly.




On initial execution, the `chromium` directory is created in the configuration directory (Fedora Linux 42: `/home/<username>/.config`). The name for the subdirectory of the default profile is `Default`:

```
~/.config/chromium/Default
```

A total of 13 LevelDB directories are created in the profile directory. An additional directory is created as soon as the first website that accesses the IndexedDB API is visited. The names of the subdirectories created here correspond to the origin of the respective page. The origin consists of the protocol, domain, and port number of the respective URL[7].[14] When using a standard protocol port, a 0 is appended instead of the assigned port number.




[6] `https://www.chromium.org/getting-involved/download-chromium/`
[7] Uniform Resource Locator

## 3.3 Procedure for Interface Analysis

The built-in Chrome DevTools of the web browser are used to generate the necessary data. These provide comprehensive tools for examining a website and the web interfaces or databases used in detail, among other things. JavaScript commands can be executed via the integrated console, allowing database data to be generated and queried directly via the corresponding APIs.

To extract the data generated in this way from LevelDB, the script `dump_leveldb.py`[9] by Alex Caithness is used.

In order to obtain results that are as unbiased and direct as possible, an empty HTML page (`index.html`) represents the origin in the test setup. This is provided locally via a Python web server using the command listed below:

```
$ python -m http.server

Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/)
```

The origin can be reached via the IP address `0.0.0.0` or `localhost` on port 8000. Transmission takes place via the HTTP protocol.

## 3.4 Web Storage

Simple key-value pairs can be stored using the Web Storage API. Web Storage is divided into *Session Storage* and *Local Storage*. Session Storage holds data for the current session and removes it when the session ends. Local Storage stores data permanently.[4]

Only strings are supported as data formats. Although numbers can also be set directly as keys or values, they are converted internally to strings during storage. Native storage of binary data is also not provided, but can be done, for example, after prior conversion to the character-based Base64 format.[4]

Web Storage is generally limited to a maximum size of 10 MiB (5 MiB Local Storage and 5 MiB Session Storage) per origin in all browsers and is therefore not suitable for storing larger amounts of data.[15]

The main functions of the API are the commands `setItem("key", "value")` to store a key-value pair, and `getItem("key")` to retrieve a previously stored value using the key passed.

### 3.4.1 Session Storage

The LevelDB database for Session Storage is located in the following directory:

```
~/.config/chromium/Default/Session Storage
```

Before working with the API, the original state of the database was examined. The following two entries were found:

| # | Key | Value |
|---|---|---|
| 1 | version | 1 |
| 2 | namespace- | |

The first entry shows the version of the Web Storage API used. The second entry was marked as deleted and contained no value. It is generally used to assign the respective page source (origin).[16]

When the empty HTML page is accessed via the address `localhost:8000`, two additional entries are automatically created in the database:

| # | Key | Value |
|---|---|---|
| 3 | next-map-id | 1 |
| 4 | namespace-<uuid>-http://localhost:8000/ | 0 |

The value of the fourth entry corresponds to the assigned origin ID, which is referred to as `map-id`. This is incremented and recorded in advance in the form of the key `next-map-id` for subsequent origin IDs.[16]

The following two entries were then created via the Web Storage API using the DevTools console:

```
sessionStorage.setItem('Shakespeare','Hamlet')
sessionStorage.setItem('Goethe','Faust')
```

This created the following two entries in the database:

| # | Key | Value |
|---|---|---|
| 5 | map-0-Shakespeare | Hamlet (UTF-16LE) |
| 6 | map-0-Goethe | Faust (UTF-16LE) |

For mapping purposes, the previously defined Origin ID is prefixed to the set keys in the form: `map-<ID>-<Key>`. Unlike other entries, the values are not encoded in UTF-8, but in UTF-16LE.[16]

When closing the browser tab or web browser, the following three entries were also generated:

| # | Key | Value |
|---|---|---|
| 7 | `namespace-<uuid>-http://localhost:8000/` | |
| 8 | `map-0-Shakespeare` | |
| 9 | `map-0-Goethe` | |

The keys with empty values are marked as deleted and are no longer returned together with the previously created and assigned entries when queried via the API. However, these and previously set entries remain completely in the database, including the set values, and can therefore be restored until the respective database file is overwritten or deleted.

### 3.4.2 Local Storage

The LevelDB database for Local Storage is located in the following directory:

```
~/.config/chromium/Default/Local Storage/leveldb
```

In its initial state, as with Session Storage, the version of the Web Storage API used could be determined in the first entry:

| # | Key | Value |
|---|---|---|
| 1 | `version` | `1` |

In contrast to Session Storage, Local Storage does not automatically generate entries for the origin when the address `localhost:8000` is accessed.

The following two entries were created via the API using the DevTools console:

```
localStorage.setItem('Homer','The Iliad')
localStorage.setItem('Dante','The Divine Comedy')
```

This generates three database entries for each API entry, which are created as illustrated below:

| Key | Value |
|---|---|
| `META:http://localhost:8000` | `08 d8 ab 90 b4 e0 d4 e4 17 10 10 (hex)` |
| `METAACCESS:http://localhost:8000` | `08 d8 ab 90 b4 e0 d4 e4 17 (hex)` |
| `_http://localhost:8000[0x00][0x01]Homer` | `[0x01]The Iliad` |

PREFIX DELIMITER ENCODING varint TIMESTAMP varint SIZE

The keys `META:<origin>` and `METAACCESS:<origin>` are generated automatically. The values of both entries are protobuf-encoded[12] and contain the creation timestamp of the entry in microseconds since January 1st, 1601. The timestamp value of the `META` key is followed by the reserved size of the corresponding entry in bytes.[16]

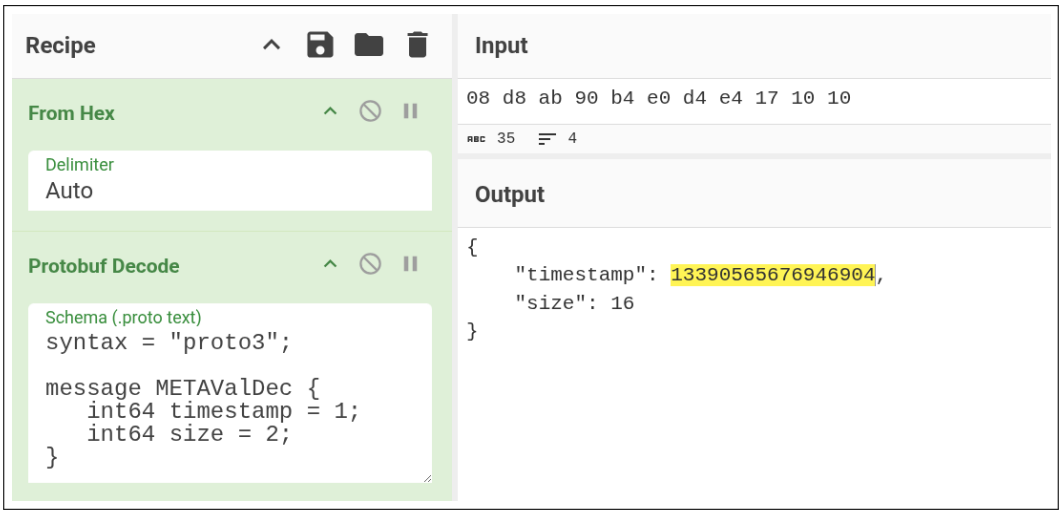



Figure 3.1: Decoding of the META value with CyberChef

The key entered via the API is prefixed with an underscore and is stored in the following format: `_<origin>0x00[0x00|0x01]<key>`.

The zero byte following the origin corresponds to a separator, while the byte preceding the key and value indicates the text encoding used. The byte `0x00` stands for UTF-16LE, while the byte `0x01` detected here corresponds to the text encoding ISO-8859-1 (Latin-1).[16]

Interestingly, the command history of DevTools is also logged with a timestamp in Local Storage and is not deleted even after closing the browser:

| Key | Value |
|---|---|
| `META:devtools://devtools` | `08 9c e9 d6 bd e0 d4 e4 17 10 9a 01 (hex)` |
| `_devtools://devtools[0x00][0x01] console-history` | `[0x01][""localStorage.setItem('Homer','The Iliad')"","localStorage.setItem('Dante','The Divine Comedy')""]` |

Unlike in Session Storage, entries in Local Storage remain permanently stored and must be explicitly deleted using the commands `removeItem("key")` or `clear()`. The first command deletes a specific entry, while the second command deletes the entire database for the respective origin.

The deletion is logged with a timestamp using the `META` key. In this case, the data also remains completely in the database and can therefore be restored until the respective database file is overwritten or deleted.

## 3.5 IndexedDB

Compared to the flat key-value structure of the Web Storage API, the object-oriented, transaction-based, and asynchronous IndexedDB API has a significantly more complex structure.

This means that multiple databases can be created under one origin, each containing several *Object Stores*. These are comparable to tables in relational databases, but do not require a fixed schema for the stored data objects. Auto-incrementing primary keys and the name-giving index are also available.

Supported data formats include strings, integers and floating point numbers, boolean and binary values, and JavaScript objects.

IndexedDB can basically access the entire mass storage of the respective system. However, web browser manufacturers currently set limits of between 50 % and 60 % of the total capacity of the mass storage per origin.[15]

This thesis can and will only give a rough overview of IndexedDB. For the sake of comparability with the Web Storage API, only character-based key-value pairs will be used. A more detailed description of the implementation can be found in the Chromium documentation[17] or, for example, in the article *IndexedDB on Chromium*[18] by Alex Caithness.

The LevelDB databases for IndexedDB are located in the following directory:

```
~/.config/chromium/Default/IndexedDB
```

Unlike Web Storage, which maintains a central database, IndexedDB creates a separate subdirectory with its own database for each origin. This strict separation of databases is intended to ensure that one origin cannot access the data of another origin.

The database directory is created when the respective origin accesses the IndexedDB API for the first time, using the following naming scheme:

```
<protocol>_<domain>_<port>.indexeddb.leveldb
```

In order to generate a database directory, two databases were first created via the IndexedDB API using the DevTools console:

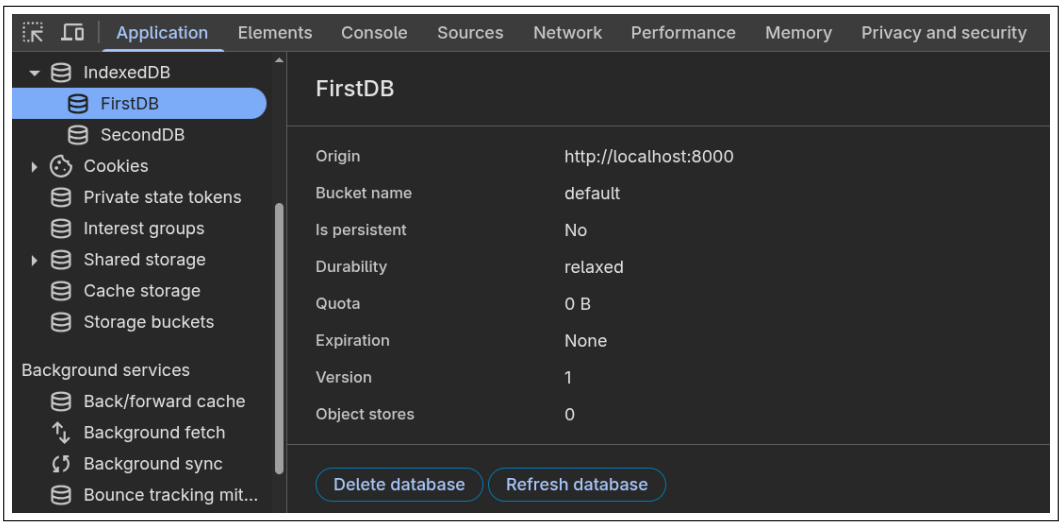
```
indexedDB.open('FirstDB').onsuccess=e=>e.target.result.close()
indexedDB.open('SecondDB').onsuccess=e=>e.target.result.close()
```

The code creates and opens the databases named `FirstDB` and `SecondDB`, respectively, and closes them immediately after creation.

The following subdirectory is created in the IndexedDB directory:

```
http_localhost_8000.indexeddb.leveldb
```

The databases that have been created and are still empty can then be viewed using DevTools, for example:




In addition to the actual entries, various metadata is stored in the database. This can be divided into global metadata and metadata for the respective database and individual object stores, which will be discussed in more detail below.

### 3.5.1 Global Metadata

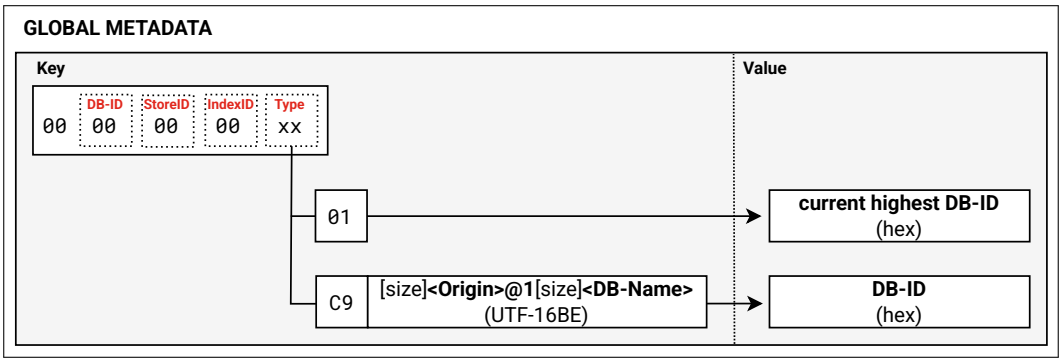
All database keys begin with a key prefix. Provided that the sizes of the generated databases, object stores, or object store indexes do not exceed the value 255, the key prefix consists of a total of four bytes.[18]

The size of these three values is encoded in the first byte and is always `0x00` if they are less than 255.[18]

The second byte represents the database ID, which is incremented by one for each newly created database. The global metadata has the database ID `0x00`.[18]

The third byte identifies the object store ID, while the fourth byte represents the index ID.[18]

The byte following the key prefix identifies the metadata type. Relevant identifiers include `0x01` and `0xC9`. The metadata type `0x01` has the currently highest database ID in its value. The metadata type `0xC9` contains the origin[8] and the database name in UTF-16BE encoding in its key, while the byte preceding the respective string indicates the reserved size. The value contains the ID for the specified database.[18]




### 3.5.2 Database Metadata

Before creating the database entries, an object store was first generated:

```
indexedDB.open('FirstDB',2).onupgradeneeded=e=>{
    e.target.result.createObjectStore('books')
    e.target.transaction.oncomplete=()=>e.target.result.close()
}
```

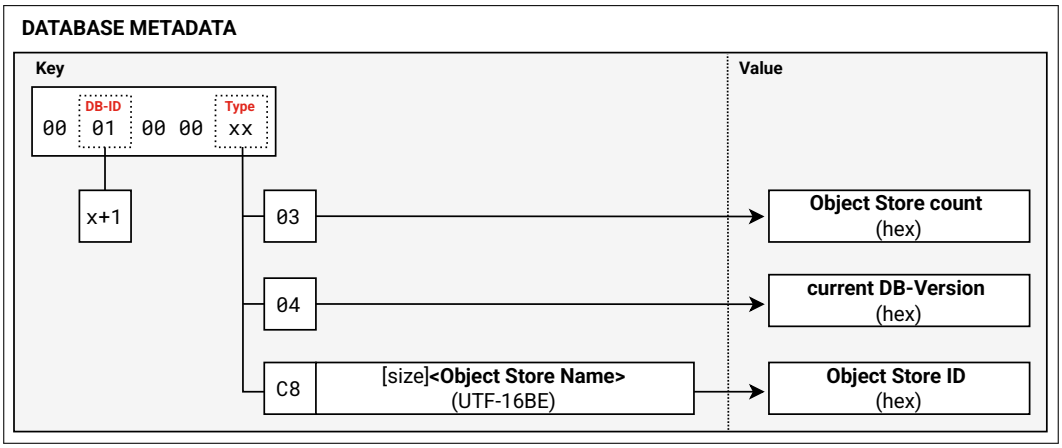
The code first opens the database `FirstDB`. When structural changes are made to the database, such as the creation of an object store in this case, the database version must be increased, which is raised to version 2 here.

The object store with the name `books` is then created. Since the Indexed DB API is transaction-based, this is carried out as part of a transaction and the database is closed at the end.

The previously created databases have the database ID `0x01` for the database `FirstDB` and the database ID `0x02` for the database `SecondDB`.

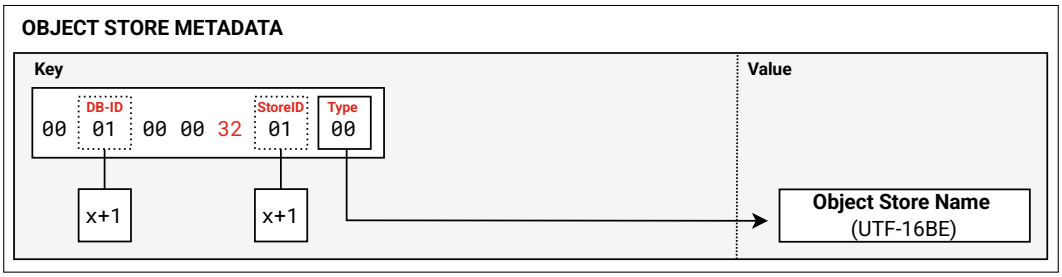
[8] For backward compatibility reasons, the origin always has the version suffix @1: `https://github.com/chromium/chromium/blob/87848c4278450ffe4159d74732660b5e65a28180/content/browser/indexed_db/instance/leveldb/backing_store.h#L653`

The relevant type identifiers include the bytes `0x03`, `0x04`, and `0xC8`. The byte `0x03` indicates the number of object stores in the respective database, and the byte `0x04` indicates the current database version. The byte `0xC8` is followed by the name of the respective object store, while the value contains the ID of the object store.[18]




### 3.5.3 Object Store Metadata

The metadata for the object store begins with the type identifier `0x32`. This is followed by the ID of the associated object store. The relevant identifier is the byte `0x00`, which contains the name of the respective object store in its value.[18]



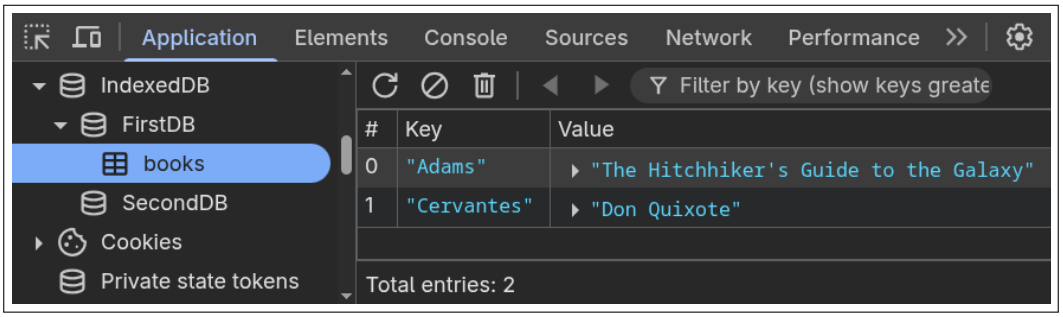

### 3.5.4 Database Entries

The database entries were generated using the following code:

```
indexedDB.open('FirstDB')
    .onsuccess=e=>
        e.target.result.transaction('books','readwrite')
            .objectStore('books')
            .add('Don Quixote','Cervantes')
            .oncomplete=()=>e.target.result.close()
```

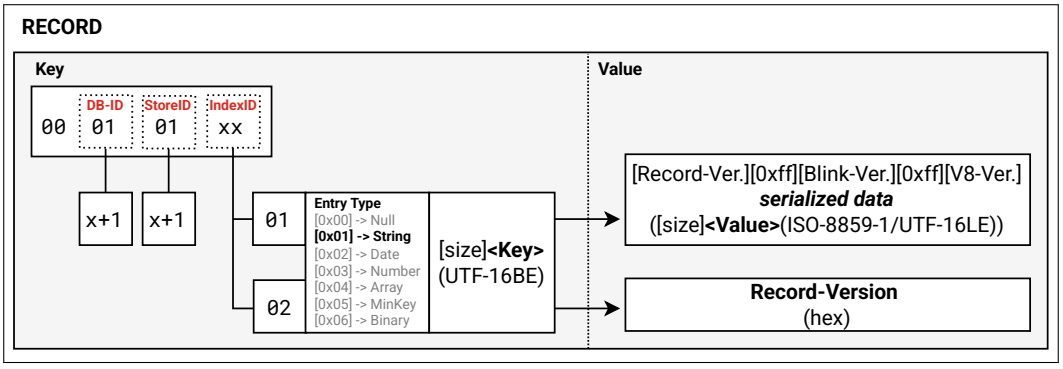
After opening the database `FirstDB`, a transaction with write access (`readwrite`) is performed. This accesses the previously created object store named `books` and adds the respective key-value pair before closing the database again.

The entries generated in this way can be viewed with DevTools:




The entries can be assigned using the database ID in combination with the object store ID. The subsequent index ID can be used to determine the set value using the `0x01` identifier or the record version using the `0x02` identifier.[18]

The record version number is incremented by one for each change and new entry, so that the order of entries and any changes can be determined chronologically.




The set key is determined using one of the seven entry types (`0x00–0x06`). This can be binary data, a number, or, as in this case, a string preceded by the identifier `0x01`. The length of the string is indicated by the preceding varint.[18]

The value entry is preceded by the record version and the version of the Blink and V8 engines used. This is followed by the serialized data. Where possible, strings are encoded in Latin-1 (ISO-8859-1), otherwise in UTF-16, with the byte order determined by the host system[9].[18] Given the common endianness conventions of

[9] `https://github.com/v8/v8/blob/5fe0aa3bc79c0a9d3ad546b79211f07105f09585/src/objects/value-serializer.cc#L321`

modern processors, this is likely to be UTF-16LE in most cases, although not without exception.

As with the Web Storage API, deleted entries via the IndexedDB API initially remain stored in the database and can be viewed and restored until the corresponding database files are deleted.

## 3.6 Miscellaneous

As soon as Chromium is used in incognito mode, entries are stored in the main memory via the Web Storage API and IndexedDB API. Data persistence is limited to the lifetime of the private session and is completely discarded when the private window is closed. Accordingly, in this case, no data is stored in the LevelDB directories described above. The Chrome DevTools command history is also not stored in this case.

To address this issue, Byeongchan Jeong, Sangjin Lee, and Jungheum Park from the Korea University School of Cybersecurity developed the script *MIC - Memory analysis of IndexedDB data on Chromium-based applications*[10], which is designed to extract relevant data from memory dumps. They presented their work in an article of the same title in the journal *Forensic Science International: Digital Investigation* in October 2024.[19]

This excursus was limited to the main methods used for local storage of application and user data via the interfaces described. As seen in section 3.2, LevelDB directories are not only created for Web Storage and IndexedDB, but also for data caching or browser extensions[11], for example, so that relevant traces can also be expected in these databases in the corresponding directories.

[10] `https://github.com/naaya17/MIC`

[11] Small software programs that are integrated directly into the web browser to extend its functionality, which can store their own data.

# Chapter 4

# Development of the Parser

This chapter describes the development of the parser for LevelDB. The purpose of the parser is to read data structures from LevelDB, interpret them, and present them in a form that is understandable to the user.

The functional and non-functional requirements for the parser are defined as part of a requirements analysis. Based on this, the concept and design are developed, with a presentation of the central components and technical decisions.

The final section on implementation explains the specific realization of the parser on a logical and technical level. This is divided into three components: the parser library (LIB), which reads and interprets the LevelDB data; the command line application (CLI) for detailed analysis of individual files; and the user interface application (GUI), which can display the data from multiple databases in a processed and searchable form in a table.

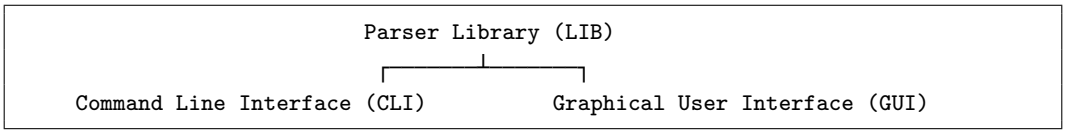
## 4.1 Requirements Analysis

The parser should be implemented for both the command line (CLI) and as a graphical user interface (GUI). It should be executable on all common operating systems (Linux, Mac, Windows); operating system-dependent code should therefore be avoided.

As a forensic application, the parser should not only extract and display the database contents in the form of key-value pairs, but also provide deleted entries and all existing metadata. The integrity of the individual data blocks should be cross-checked using the existing checksums (CRC32C) and the result should be explicitly output in order to be able to detect subsequent data changes.

## 4.2 Concept and Design

The parser code is moved to a separate library along with other helper functions to ensure reusability when implementing the applications and to avoid redundancy in the code:




The **library** is divided into individual modules according to their function:

```
lib
 ├── ldb_parser        (.ldb)
 ├── log_parser        (.log)
 ├── manifest_parser   (MANIFEST)
 ├── log_text_parser   (LOG)
 └── utils             (helper functions)
```

The **command line application** is implemented entirely in the Rust programming language. The application can accept individual binary files (.ldb, .log, MANIFEST) from LevelDB, parse them, and output the corresponding results. The key-value pairs are output as CSV (Comma-Separated Values) to enable further reuse of the data:

```
"seq","state","key","value"
"1","Live","Mozart","Eine kleine Nachtmusik"
"2","Live","Bach","Air"
"3","Live","Vivaldi","Le quattro stagioni"
```

The `-a` (all) option enables detailed output, including metadata and the hex byte representation of all entries:

```
########## [ Block 2 (Offset: 50)] ############
------------------- Header -------------------
CRC32C: DC3ADC4D (verified)
Data-Length: 22 Bytes
Record-Type: 1 (Full)
///////////////// Batch Header ////////////////
Seq: 2
Records: 1
****************** Record 1 ******************
Seq: 2, State: 1 (Live)
Key (Offset: 119, Size: 4): '\x42\x61\x63\x68'
Val (Offset: 124, Size: 3): '\x41\x69\x72'
```

The **graphical user interface application** is implemented using the open-source Tauri framework. This allows Rust to be used in the logic layer (backend) and web technologies (HTML, CSS, JS) in the presentation layer (frontend) via the WebView of the respective operating system.

The content is presented in tabular form. This is achieved using the open source library AG Grid Community, which enables performant presentation and searching of large data sets.

In addition to individual files, the application can also accept folders containing multiple LevelDB databases in order to recursively process all files contained therein and display the contents in a summarized form.

The records of the .ldb and .log files are displayed in a shared tab. The manifest contents and database log entries are displayed in separate tabs. All content is fully searchable. Checksum discrepancies that indicate data changes or corruption are highlighted in color for better visibility.

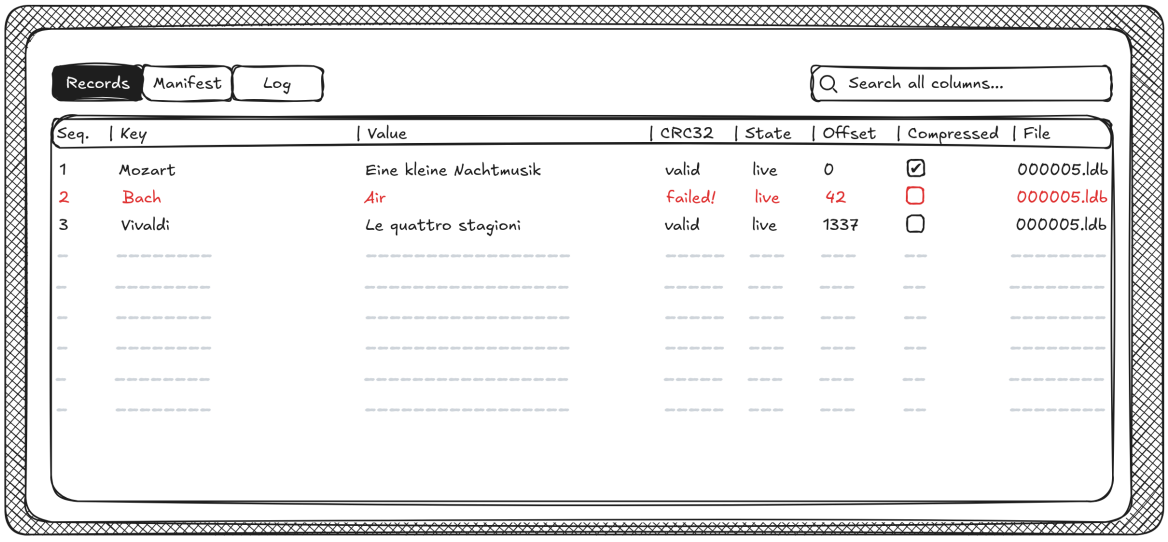



Figure 4.1: GUI sketch (created with: excalidraw.com)

## 4.3 Implementation

The source code is documented and published on the GitHub platform under the free MIT license.[1] The license allows free use and modification of the source code, provided the author is named. Among other things, the platform allows users to view the source code and track its historical development. In addition to the source code, also the compiled applications are provided on the platform.

### 4.3.1 Library (LIB)

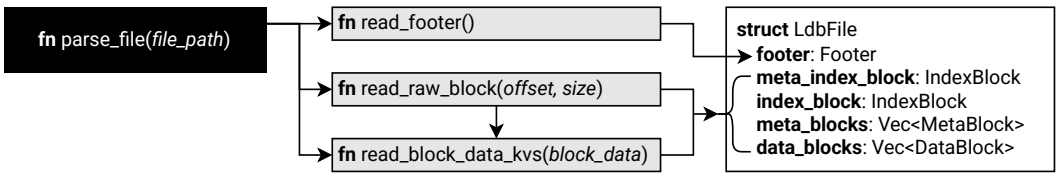
The library `lib.rs` is separated into the following modules: `log_parser.rs`, `ldb_parser.rs`, `manifest_parser.rs`, `log_text_parser.rs` and `utils.rs`.

#### `ldb_parser.rs`

This module is used to parse .ldb files. The primary function, `parse_file()`, accepts the file path of the file to be parsed and outputs a struct containing the processed contents of the file in the form of the footer, meta index block, index block, meta blocks, and data blocks.

The respective blocks are read and processed using the helper functions `read_footer()` for reading the footer, `read_raw_block()` for reading the raw blocks, and for reading and processing the key-value pairs from the raw blocks via the `read_block_data_kvs()` function.

When compressed blocks are detected, they are decompressed using the external libraries Snap[2] or Zstd[3].

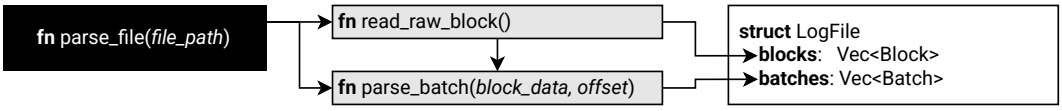



#### `log_parser.rs`

This module parses .log files. The primary function `parse_file()` is passed the path of the file to be parsed, which outputs a struct containing the processed contents of the file in the form of both raw blocks and fully assembled partial blocks (batches).

[1] `https://github.com/huebicode/leveldb-parser` [20]
[2] `https://crates.io/crates/snap`
[3] `https://crates.io/crates/zstd`

The raw blocks are read in using the helper function `read_raw_block()`. The program then checks whether the block is a complete block or a partial block. If it is a complete block, it is read in and processed using the helper function `parse_batch()`. If it is a partial block, it is first assembled before being passed to the helper function.

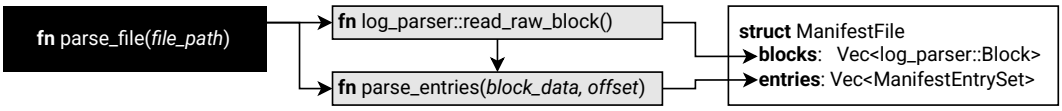



### `manifest_parser.rs`

This module is responsible for parsing manifest files. The function `parse_file()` accepts the file path of the file to be parsed and outputs a struct that contains not only the raw blocks but also the assembled blocks in the form of individual manifest entries (ManifestEntrySet).

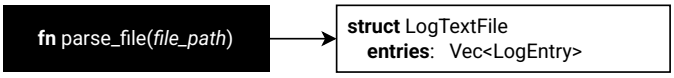
Since the data structure of the manifest files is based on the log format, the helper function `read_raw_block()` of the log parser module (`log_parser.rs`) can be reused to read the raw blocks.

The individual tags are then processed using the `parse_entries()` function. Any partial blocks are assembled beforehand.




### `log_text_parser.rs`

The module accepts `LOG`/`LOG.old` files, reads the entries in the text files line by line, and stores the timestamp, thread ID, and message of each entry in a struct.




### `utils.rs`

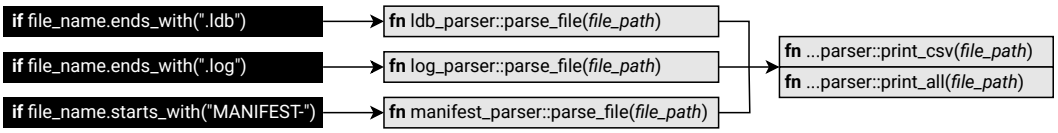
The utility module contains various helper functions that can be used across the main modules.

Among other things, it contains functions for unmasking and verifying the CRC checksum, as well as decoding the varint format and internal keys.

### 4.3.2 Command-Line Application (CLI)

The command line application imports the modules `ldb_parser`, `log_parser`, and `manifest_parser` from the library.

During file passing, the program checks the beginning and ending of the file name in order to assign it to the respective parser module and extract the contents using the corresponding parser function `parse_file()`.




The standard output is in CSV format. For this purpose, the parser modules were each extended with the function `print_csv()`:

```
alex@fedora:~$ leveldb-parser-cli 000003.log
"seq","state","key","value"
"1","Live","Mozart","Eine kleine Nachtmusik"
"2","Live","Vivaldi","Le quattro stagioni"
"3","Live","Bach","Air"
"4","Deleted","Mozart",""
"5","Live","Bach","Das wohltemperierte Klavier"
```

Figure 4.2: Example output as CSV

When the `-a` option is set, a complete output is generated. For this purpose, the function `print_all()` has been implemented in the respective modules:

```
alex@fedora:~$ leveldb-parser-cli 000003.log -a

########## [ Block 3 (Offset: 98)] ############
------------------- Header -------------------
CRC32C: DC3ADC4D (verified)
Data-Length: 22 Bytes
Record-Type: 1 (Full)

//////////////// Batch Header ////////////////
Seq: 3
Records: 1

****************** Record 1 ******************
Seq: 3, State: 1 (Live)
Key (Offset: 119, Size: 4): '\x42\x61\x63\x68'
Val (Offset: 124, Size: 3): '\x41\x69\x72'
```

Figure 4.3: Example output of the third block with the option set: -a (all)

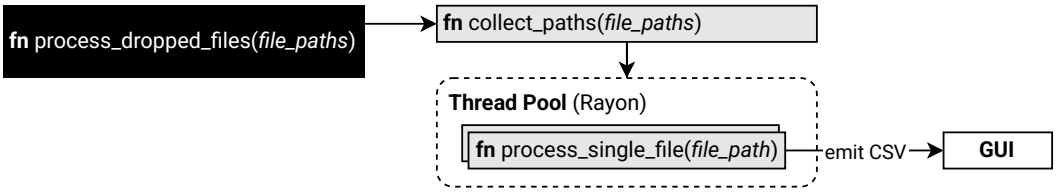
### 4.3.3 Graphical User Interface Application (GUI)

The user interface application imports the modules `ldb_parser`, `log_parser`, and `manifest_parser` from the library in the logic layer. Furthermore, the external library Rayon[4] is used, which enables parallel processing of individual files via a thread pool.

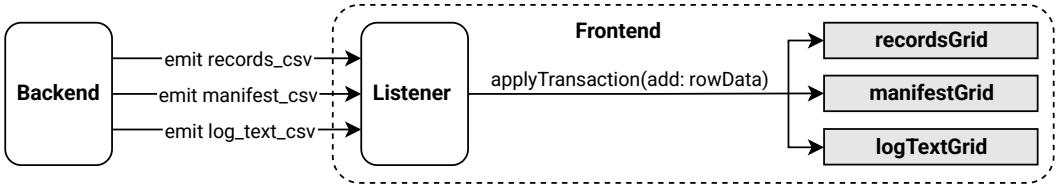
Files and folders can be transferred to the application either individually or in bulk. Processing takes place via the `process_dropped_files()` function. Upon transfer, the paths are first collected via the `collect_paths()` function and assigned to the respective file. Folders are traversed recursively.

The individual files are then processed in parallel via a thread pool using the `process_single_file()` function. As with the command line application, the `parse_file()` function of the respective parser module is used for each file.

The extracted content is transferred to the GUI for data display in CSV format, for which the parser modules have been extended with the function `csv_string()`.




The library `ag-grid-community.js` is used in the presentation layer to display the CSV data provided by the logic layer. The three tables generated are filled with the corresponding data.




The files are passed by dragging the directories or individual files onto the application window using drag-and-drop. When additional files are passed, the existing data is cumulated with the new data.

[4] `https://crates.io/crates/rayon`

The table view is divided into three tabs: Records, Manifest, and Log. The Records tab contains the data extracted from the `.log` and `.ldb` files. The Manifest tab displays the manifest files, while the Log tab displays the contents of the `LOG` and `LOG.old` files.

Next to the tabs is a search field that allows searching across all columns of the currently displayed table. Additionally, buttons have been implemented to reset all filters, export the currently displayed table as a CSV file, and discard all tables to enable restarting the data analysis process.

The individual table columns can be sorted by clicking on the respective column header and searched and filtered separately using a filter icon.


Figure 4.4: Header bar of the user interface

As soon as the calculated checksum of a block does not match the stored checksum, the font of the respective table row is highlighted in red to indicate possible data corruption/alteration. Deleted entries are highlighted in gray. If data is stored in compressed data blocks, this is indicated by a check mark in the Compressed column.

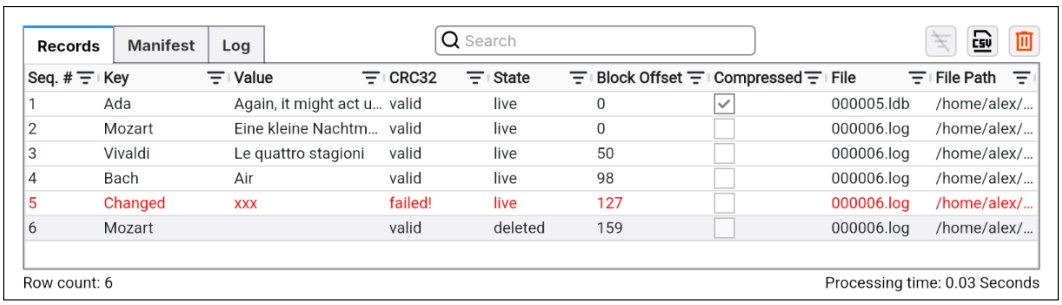



Figure 4.5: Table view of the tab: Records

LevelDB theoretically allows individual entries with a size of several gigabytes to be stored. In order to ensure high-performance display of the table even with many large entries, values longer than 300 characters are truncated. The truncation is indicated by specifying the remaining characters at the end of the respective entry in the form *[+n Chars]*. The complete searchability of the data records is retained even when an entry is truncated.

If needed, the complete entry can be opened in a separate pop-up window by double-clicking on it. When searching, the location being searched for is highlighted.

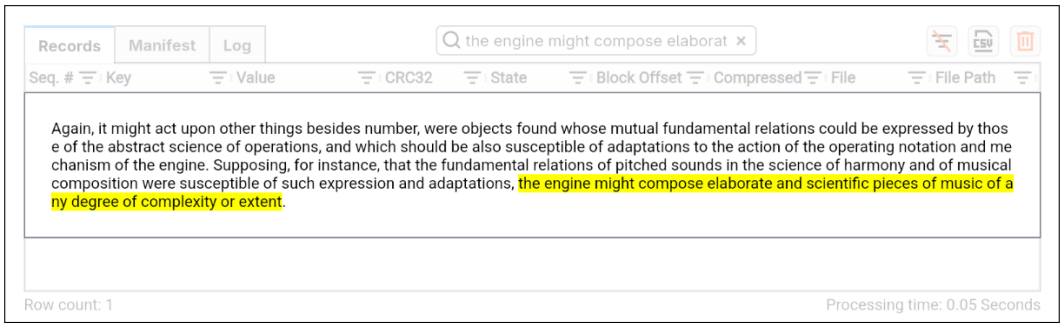



Figure 4.6: Display of an entry after double-clicking with search marker

The Manifest tab displays the contents of all manifest files in the form of tags and their values. As with the Records tab, invalid block checksums are indicated by red text.

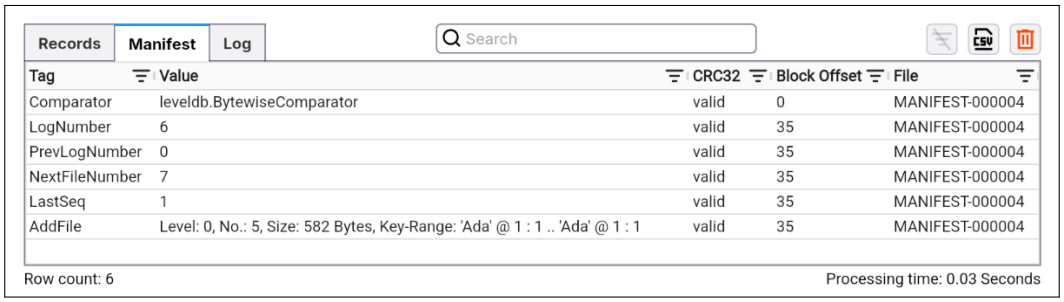



Figure 4.7: Table view of the tab: Manifest

The contents of the text logs are displayed in the Log tab. The respective entries are divided into the columns Date, ThreadID, and Message.

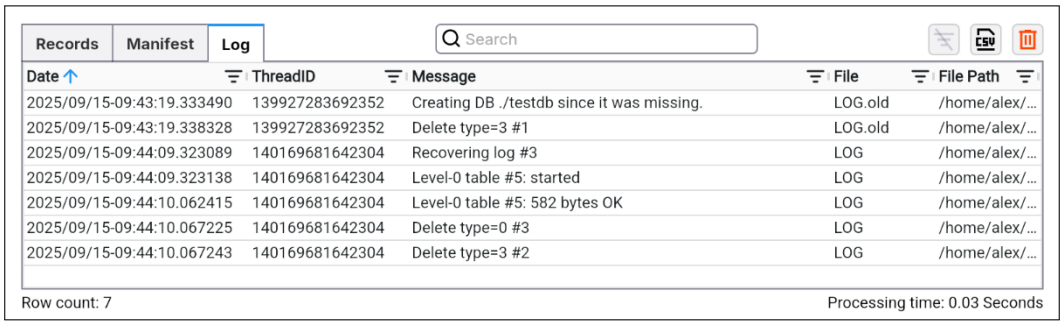



Figure 4.8: Table view of the tab: Log

# Chapter 5

# Verification and Validation

In this chapter, the developed GUI parser will be subjected to various tests to check its functionality, performance, and resource consumption.

The tests are performed on an older and a more modern system with different operating systems in order to cover as broad a test spectrum as possible:

- **System 1** – Intel i5-1130G7 (1.80 GHz, 4 processor cores, release: 2020), 16 GB RAM (DDR4-4266), Fedora Linux 42 (6.14.3-300.fc42.x86_64)
- **System 2** – AMD Ryzen 9 7940HS (4 GHz, 8 processor cores, release: 2023), 64 GB RAM (DDR5-4800), Windows 10 Pro (22H2, 19045.6216)

Three different-sized data sets are used for the tests:

- **Small data set** – Five entries with a maximum of 50 characters and one deleted, one updated, and one manipulated entry.
- **Medium data set** – Three entries, which are the works *Les Misérables*[1] and *War and Peace*[2] with over three million characters each, and *Don Quixote*[3] with over two million characters.
- **Large data set** – Real data generated after years of personal browser use in the web browser's IndexedDB directory. The directory comprises 89 databases with a total of 589,093 entries. The completeness of this data set is checked cursorively using randomly selected entries.

[1] `https://www.gutenberg.org/ebooks/135`
[2] `https://www.gutenberg.org/ebooks/2600`
[3] `https://www.gutenberg.org/ebooks/996`

## 5.1 Functional Tests

Functional tests include checking the general functionality of the application. The tests cover the complete processing and assignment of data to their respective tabs, the functionality of the buttons and status bar, and the complete display of values when double-clicking in a separate window, including the search marker. Further tests include the filter and sort function of the respective table columns as well as the highlighting of deleted entries and manipulated data.

| | System 1 | System 2 |
|---|---|---|
| Data completely and correctly assigned | ✓ | ✓ |
| Button functionality: | | |
| Search | ✓ | ✓ |
| Filter-Reset | ✓ | ✓ |
| CSV-Export | ✓ | ✓ |
| Table-Reset | ✓ | ✓ |
| Table buttons functionality: | | |
| Column sorting | ✓ | ✓ |
| Column filtering | ✓ | ✓ |
| Status bar functionality: | | |
| Row-Count | ✓ | ✓ |
| Processing-Time | ✓ | ✓ |
| Value display complete on double-click | ✓ | ✓ |
| Search marker in the value display | ✓ | ✓ |
| Highlighting manipulated blocks | ✓ | ✓ |
| Highlighting deleted entries | ✓ | ✓ |

All tests were successfully performed on both systems with all three data sets. The data outputs and displays were identical and complete. Searched words or phrases were highlighted in a separate pop-up window. However, for long entries, the highlighted hits must be searched for by manually scrolling. An additional button would be useful here to efficiently move through the hits.

## 5.2 Performance Tests and Resource Consumption

This section measures the performance and resource consumption of the GUI parser. For this purpose, temporary helper functions were implemented that record not only the processing time but also the time required for the search and for displaying the complete value in a pop-up window. To measure the pop-up display of the medium-sized data set, the largest entry (Les Misérables) was selected in each case.

The *System Monitor* was used on the Linux system (System 1) and the *Task Manager* on the Windows system (System 2) to measure RAM requirements.

The parser uses the system's own WebView. On the Linux system, this is WebKit, while Windows uses WebView2 (Edge/Chromium). The memory consumption of the application and the separate WebView processes was combined.

After launching the application, a total RAM consumption of 137 MB on the Linux system and 95 MB on the Windows system was observed during idle time. The further results regarding consumption and performance using the test data can be summarized as follows:

| | System 1 – Data set: | | | System 2 – Data set: | | |
|---|---|---|---|---|---|---|
| | small | medium | large | small | medium | large |
| RAM | 154 MB | 421 MB | 7.5 GB | 126 MB | 388 MB | 4.1 GB |
| Processing time | 0.04 s | 0.49 s | 14.2 s | 0.02 s | 0.46 s | 13.7 s |
| Search | 0.33 s | 0.38 s | 4.5 s | 0.32 s | 0.36 s | 3.3 s |
| Pop-up display | 0.03 s | 1.08 s | - | 0.02 s | 0.66 s | - |

On the more powerful Windows system, loading and displaying the large entry in the pop-up window was noticeably faster for the medium data set. Here, it would be useful to implement a loading indicator to show when large amounts of data are being loaded. Searching the large data set was also faster. Apart from that, there were no noticeable differences in performance between the two systems.

Memory consumption is significantly lower on the Windows system, especially with the large data set, which indicates a more efficient integration of the WebView. For very large data sets, it would be an option to transfer the data to mass storage, but this would result in a loss of processing time and search performance.

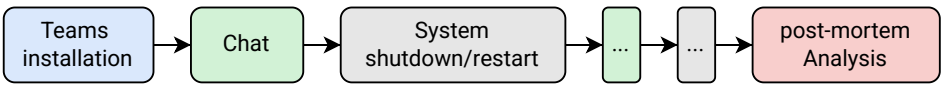
# Chapter 6

# Use Case: Microsoft Teams

The LevelDB parser that has been created will be presented in this chapter using a real-world example. The Microsoft Teams application, which provides chat and video conferencing functionalities, among other things, was selected for this purpose. Since its introduction in 2017, the application has seen steady user growth and, according to the latest figures published by Microsoft from 2023, has reached around 320 million monthly active users.[1] With this year's discontinuation of the company's own similarly positioned application Skype and the option of automatic migration to Teams, further user growth is expected.[2]

Teams uses the LevelDB database for data persistence, which is indirectly used via web APIs such as Local Storage and IndexedDB.[3, pp. 87 sqq.] For this example, a chat conversation between two users is simulated, in which images and files are sent in addition to text.

The examined system should be shut down and restarted several times in between to simulate several days of application usage and to exclude temporarily stored data as far as possible:




[1] https://www.microsoft.com/en-us/investor/events/fy-2024/earnings-fy-2024-q1

[2] https://support.microsoft.com/en-us/office/moving-from-skype-to-microsoft-teams-free-3c0caa26-d9db-4179-bcb3-930ae2c87570

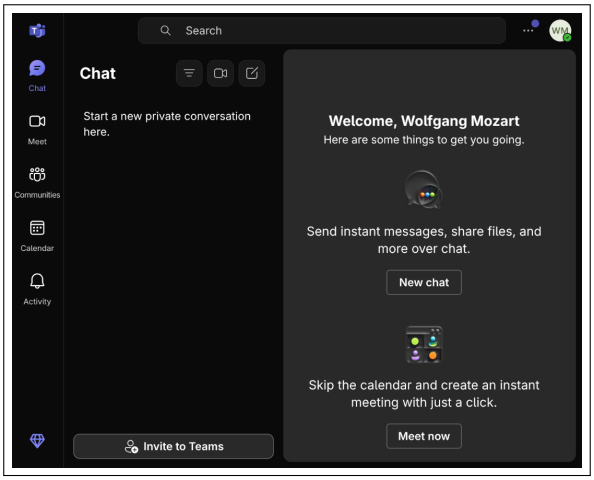
After completing the entire chat run and shutting down the system, the parser created is used in the post-mortem analysis to check whether and where the chat data can be found in order to extract and process relevant information as completely as possible in a subsequent step.

## 6.1 Preparation

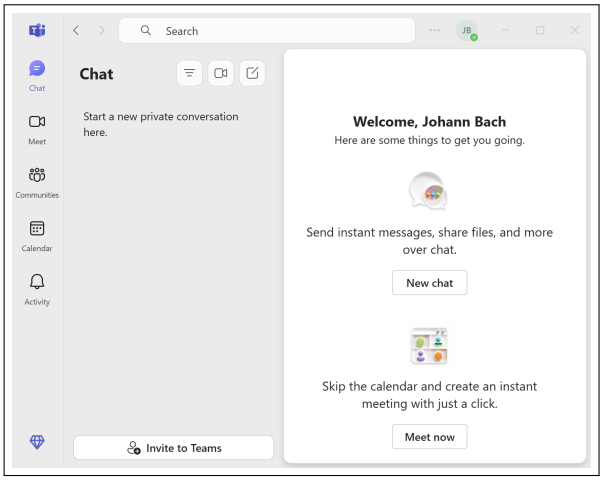
Microsoft Teams is used by the chat partners on the systems described in chapter 5. Chat partner 1 uses the Linux system on which the application runs in the Vivaldi web browser, while chat partner 2 uses the Windows system under examination. For this purpose, the application was downloaded from the official source[3] and installed.

Two new email addresses were set up for this purpose, each of which was used to create a Microsoft account that is used to log in to Teams:

| | Chat partner 1 | Chat partner 2 |
|---|---|---|
| System | Linux (System 1) | Windows (System 2) |
| E-Mail | w.a.mozart156@web.de | j.s.bach385@web.de |
| Teams | Vivaldi Webbrowser, https://teams.microsoft.com/v2/ | Application, v.25212.2204.3869.2204 |




(a) Teams on System 1

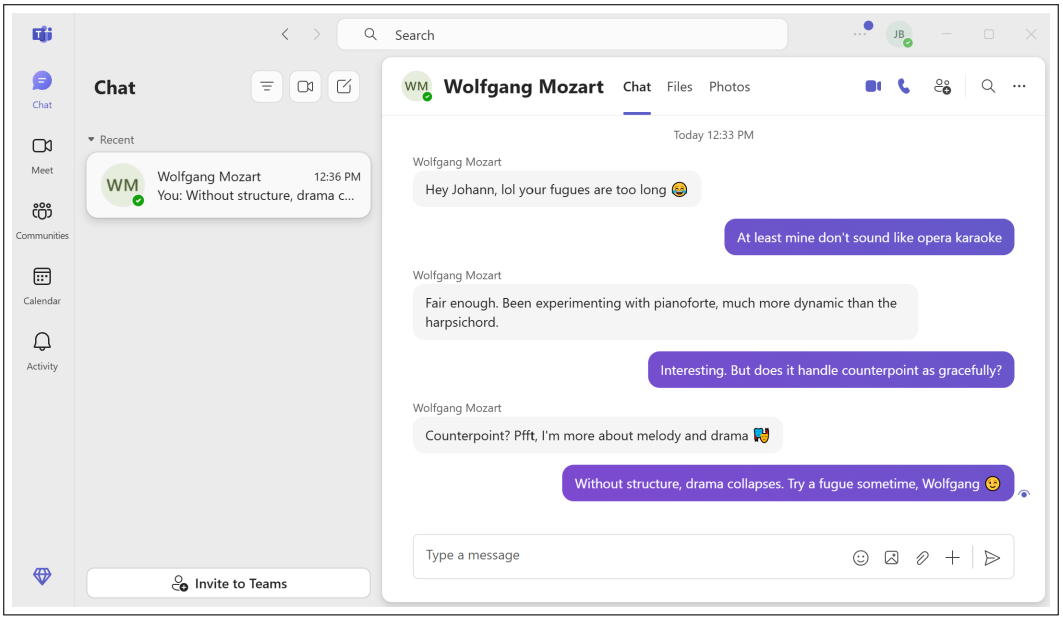



(b) Teams on System 2

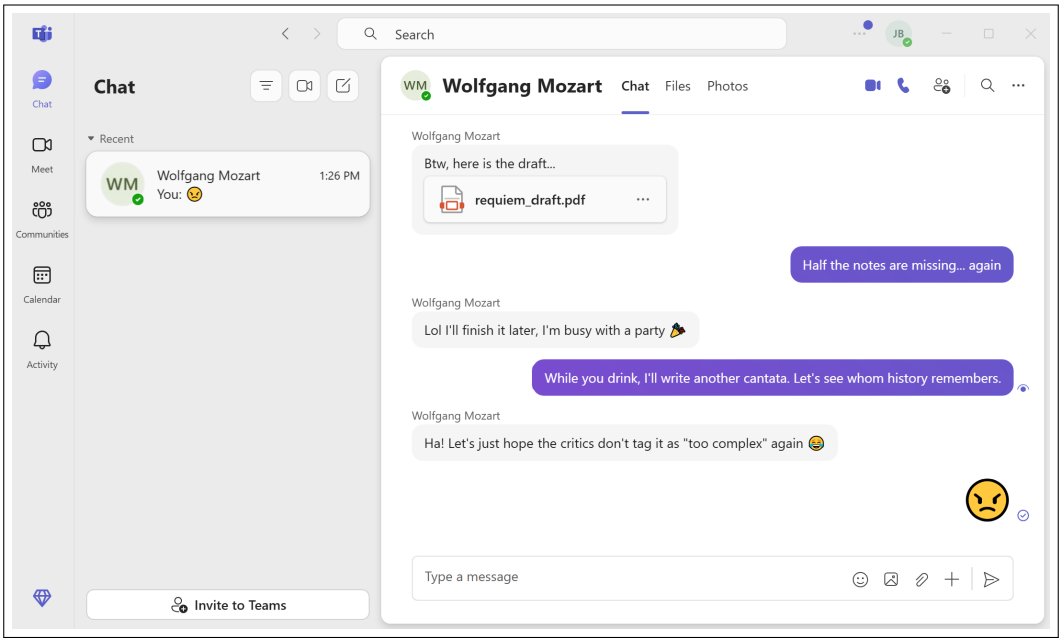
[3] https://www.microsoft.com/en-us/microsoft-teams/download-app

## 6.2 Execution

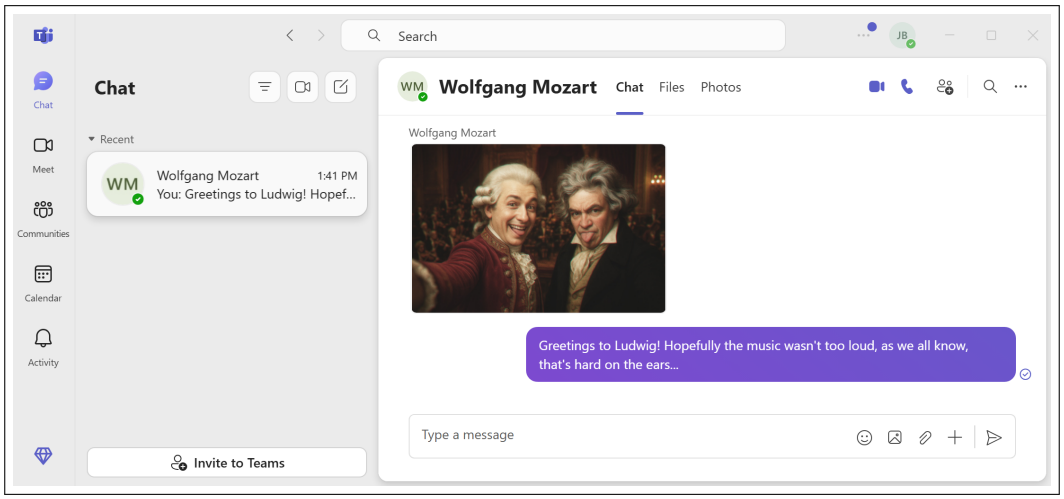
After the preparations, the invitation link was sent from the second chat partner to the first chat partner, and the first chat was conducted with a total of six messages:



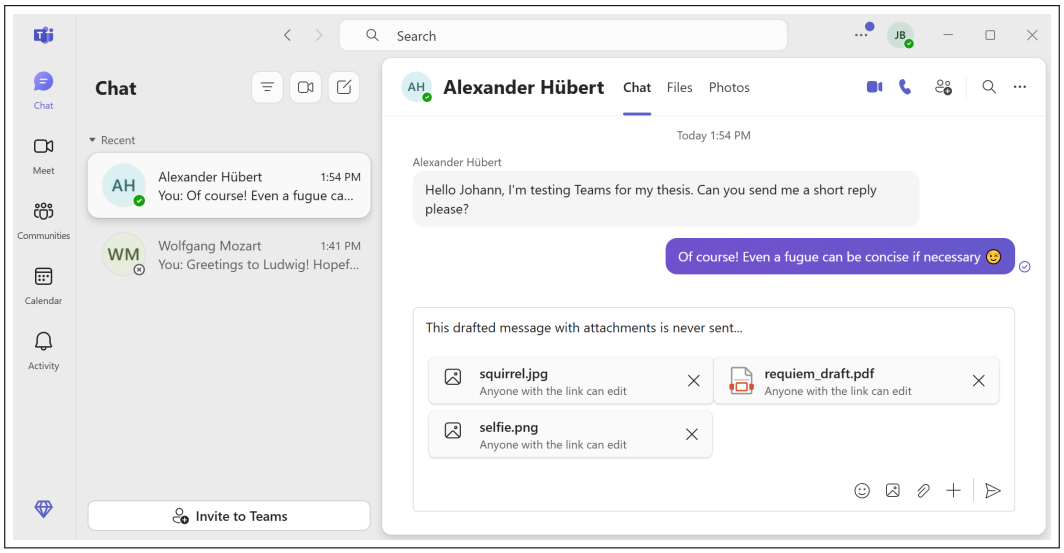

The shutdown/restart of the system was followed by a second chat with six further messages, one of which contained an attachment in the form of a PDF file:



Another shutdown and restart of the system was followed by the third chat with a directly sent image and a message:




Furthermore, two messages were generated with a new chat contact:




A message with three attachments was then created but not sent in order to test the recovery of draft messages. Another draft message was created in the chat window of the other contact.

Finally, the Windows system (System 2) of chat user No. 2 that was to be examined was shut down and the associated mass storage device in the form of an SSD[4] was removed for analysis.

[4]Solid-State Drive

## 6.3 Analysis

The analysis is performed on a Linux system (Chapter 5, System 1). For this purpose, the previously removed SSD was connected to the system in read-only mode.

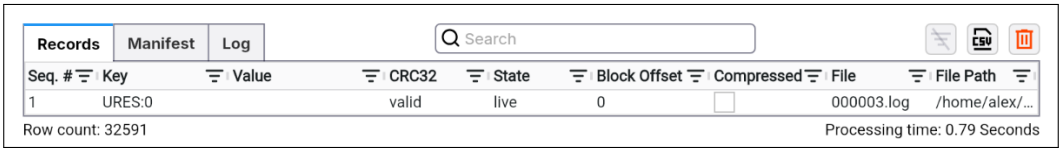
Teams uses the following path to store user data[3, p. 89]:

```
C:\Users\<User>\AppData\Local\Packages\MSTeams_8wekyb3d8bbwe
```

A search for the characteristic manifest file revealed that the application uses a total of 31 LevelDB databases in the user directory. All databases were stored in the following three main subdirectories, which were divided into further subdirectories:

```
…\MSTeams_8wekyb3d8bbwe\LocalCache\Microsoft\MSTeams\EBWebView\Default
…\MSTeams_8wekyb3d8bbwe\LocalCache\Microsoft\MSTeams\EBWebView\WV2Profile_tfl
…\MSTeams_8wekyb3d8bbwe\LocalCache\Microsoft\MSTeams\EBWebView\WV2Profile_tfw
```

To determine the relevant databases, the complete directory `MSTeams_8wekyb3d8bbwe` was passed to the parser, which searched the directories recursively and aggregated the database data:




A total of 32,591 entries were found. A search for the sent chat messages revealed that all messages were located in databases in the `WV2Profile_tfl` subdirectory. With the exception of the unsent draft messages, all messages could be found in the **IndexedDB** database in a correspondingly serialized form:

```
…\WV2Profile_tfl\IndexedDB\https_teams.live.com_0.indexeddb.leveldb
```

{"conversationId":"19:uni01_vl6knbyum5ca36dmm36w2omnr6f42euhfe4jps4ihqcdu75r6vdq@thread.v2","replyChainId":"1757586798212","latestDeliveryTime":1757586798212,"parentMessageVersion":1757586798212,"messageMap":{"8:live:.cid.abb597a863cd8a2e_2041219645654940609":{"id":"1757586798212","callId":"undefined","dedupeKey":"8:live:.cid.abb597a863cd8a2e_2041219645654940609","conversationId":"19:uni01_vl6knbyum5ca36dmm36w2omnr6f42euhfe4jps4ihqcdu75r6vdq@thread.v2","callLogsOwnerId":"undefined","parentMessageId":"1757586798212","originalParentMessageId":"undefined","clientMessageId":"2041219645654940609","sequenceId":59,"version":"1757586798212","searchKey":"19:uni01_vl6knbyum5ca36dmm36w2omnr6f42euhfe4jps4ihqcdu75r6vdq@thread.v2_1757586798212","type":"Message","contentType":"Text","activityType":"","messageType":"RichText/Html","threadType":"","postType":"undefined","clientLieResolvedTime":"undefined","deletionInfo":"undefined","clientArrivalTime":1757586969717,"originalArrivalTime":1757586798212,"prioritizeImDisplayName":false,"imDisplayName":"Wolfgang Mozart","fromDisplayNameInToken":"undefined","fromFamilyNameInToken":"undefined","fromGivenNameInToken":"undefined","creator":"8:live:.cid.abb597a863cd8a2e","content":" Hey Johann, lol your fugues are too long 😂 ","contentHash":"741753140","source":1,"properties":{"mentions":"[]","cards":"[]","links":"[]","files":"[]","formatVariant":"TEAMS","languageStamp":"languages=en:100;fr:80;nl:73;length:43;&detector=Bling"},"skypeGuid":"undefined","translation":"undefined","isSanitized":true,"isConversationLastMessage":false,"isConversationLastMessageSanitized":false,"originalNonLieMessage":"undefined","annotationsSummary":"undefined","hasAnnotated":"undefined","state":1,"dlpData":"undefined","inlineImages":"undefined","isSentByCurrentUser":false,"draftDetails":"undefined","pinnedTime":"undefined","crossPostData":"undefined","sendPipelineStatus":null,"streamingMetadata":"undefined","skypeeditedid":"undefined"}},"messageSearchKeys":[0,"19:uni01_vl6knbyum5ca36dmm36w2omnr6f42euhfe4jps4ihqcdu75r6vdq@thread.v2_1757586798212"],"lastUpdatedBy":1,"consumptionHorizon":"undefined","consumptionHorizonBookmark":"undefined","consumptionHorizonBookmarkVersion":"undefined","followState":"undefined","rcmlastUpdatedTime":"undefined","rcmSource":"undefined","isRootSaved":"undefined","savedL2Ids":"undefined"}

In this case, the emojis are not displayed correctly because the entries are encoded in UTF-16 (see chapter 3.4 [Web Storage] and chapter 3.5 [IndexedDB]), while the parser uses UTF-8.

The two message drafts were found in **Local Storage**:

```
…\WV2Profile_tfl\Local Storage\leveldb
```


{"id":"draft.19:uni01_vl6knbyum5ca36dmm36w2omnr6f42euhfe4jps4ihqcdu75r6vdq@thread.v2","message":{"content":" This is another drafted message for Wolfgang... ","clientMessageId":"draft-656f79cb-c536-44de-b645-194532bf3664","replyChainId":null,"originalArrivalTime":null,"messageType":"RichText/Html","importance":"Standard","subject":"","files":[],"meeting":null,"deeplinks":[],"computed":{"urlPreview":null,"mentions":[],"onBehalfOf":[],"cards":[],"amsReferences":[],"offlineAssets":[],"isExpanded":false,"embeddedLinks":[],"appIdsToInstall":[],"quotedMessages":[]},"policyViolation":null,"scheduledSendTime":null,"forwardMessageId":null,"forwardMessageOriginalConvId":null,"forwardMessageRecipients":[],"forwardedMessageContext":null,"forwardAppIdsToInstall":[],"forwardMessageCardContents":null,"forwardPrivateReply":false,"isLoading":null,"lastEditTimestamp":"1757591882683","draftCreationTime":"1757591871379","postType":null,"title":null,"replyPermission":null,"fluidUrl":null,"fluidUrlPermissionInfo":null,"fluidProperties":null,"crossPostChannels":[],"crossPostChannelsIncludesPrivateOrSharedChannel":false,"crossPostId":null,"sendPostAsEmail":null,"composeCopilotData":null,"selectedAppId":null,"selectedAppStatus":null,"selectedAppInstallationState":null,"selectedCommandText":null,"shareToL1":null,"sharedToMessageId":null}}

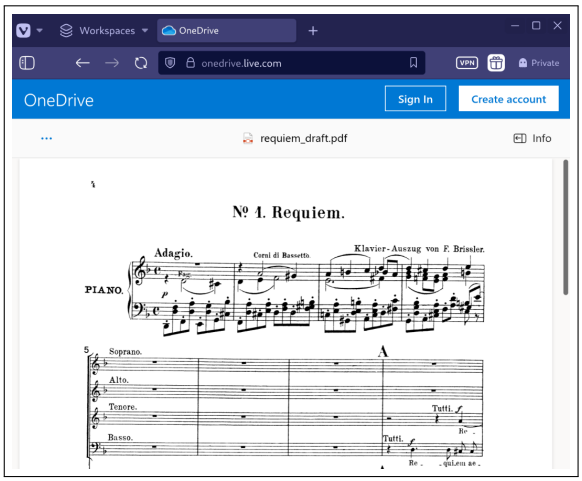

The last message sent was also stored in Local Storage, where it was available in unserialized form compared to its IndexedDB entry:


{"id":"draft.19:uni01_5tne7lo5y3kmwt2j2glzztgetp3z4otyalwkgq2ou4l5zmnj53yq@thread.v2","message":{"content":" Of course! Even a fugue can be concise if necessary 😉 ","clientMessageId":"draft-34281da6-ba7c-4bac-84e0-dc2c834ba8b2","replyChainId":null,"originalArrivalTime":null,"messageType":"RichText/Html","importance":"Standard","subject":"","files":[],"meeting":null,"deeplinks":[],"computed":{"urlPreview":null,"mentions":[],"onBehalfOf":[],"cards":[],"amsReferences":[],"offlineAssets":[],"isExpanded":false,"embeddedLinks":[],"appIdsToInstall":[],"quotedMessages":[]},"policyViolation":null,"scheduledSendTime":null,"forwardMessageId":null,"forwardMessageOriginalConvId":null,"forwardMessageRecipients":[],"forwardedMessageContext":null,"forwardAppIdsToInstall":[],"forwardMessageCardContents":null,"forwardPrivateReply":false,"isLoading":null,"lastEditTimestamp":"1757591842483","draftCreationTime":"1757591842483","postType":null,"title":null,"replyPermission":null,"fluidUrl":null,"fluidUrlPermissionInfo":null,"fluidProperties":null,"crossPostChannels":[],"crossPostChannelsIncludesPrivateOrSharedChannel":false,"crossPostId":null,"sendPostAsEmail":null,"composeCopilotData":null,"selectedAppId":null,"selectedAppStatus":null,"selectedAppInstallationState":null,"selectedCommandText":null,"shareToL1":null,"sharedToMessageId":null}}

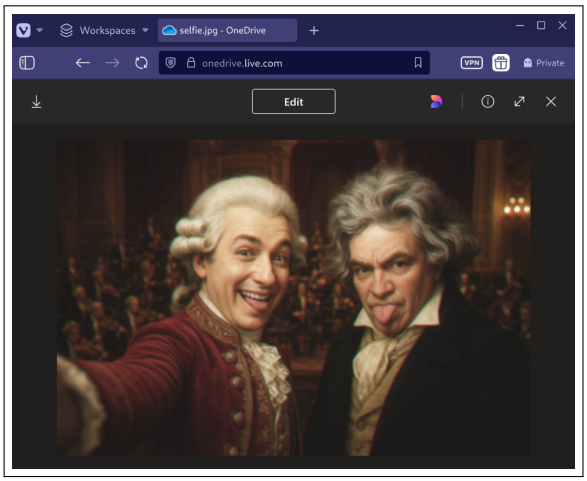

The information about the attachments sent could also be found in the respective chat entries:


{"conversationId":"19:uni01_vl6knbyum5ca36dmm36w2omnr6f42euhfe4jps4ihqcdu75r6vdq@thread.v2","replyChainId":"1757589802321","latestDeliveryTime":1757589802321,"parentMessageVersion":1757589802321,"messageMap":{"8:live:.cid.abb597a863cd8a2e_4517819635588070 79":{"id":"1757589802321","callId":"undefined","dedupeKey":"8:live:.cid.abb597a863cd8a2e_4517819635588070 79","conversationId":"19:uni01_vl6knbyum5ca36dmm36w2omnr6f42euhfe4jps4ihqcdu75r6vdq@thread.v2","callLogsOwnerId":"undefined","parentMessageId":"1757589802321","originalParentMessageId":"undefined","clientMessageId":"4517819635588070 79","sequenceId":65,"version":"1757589802321","searchKey":"19:uni01_vl6knbyum5ca36dmm36w2omnr6f42euhfe4jps4ihqcdu75r6vdq@thread.v2_1757589802321","type":"Message","contentType":"Text","activityType":"","messageType":"RichText/Html","threadType":"chat","postType":"undefined","clientLieResolvedTime":"undefined","deletionInfo":"undefined","clientArrivalTime":1757589974067,"originalArrivalTime":1757589802321,"prioritizeImDisplayName":false,"imDisplayName":"Wolfgang Mozart","fromDisplayNameInToken":"undefined","fromFamilyNameInToken":"undefined","fromGivenNameInToken":"undefined","creator":"8:live:.cid.abb597a863cd8a2e","content":" Btw, here is the draft... ","contentHash":"2952949476","source":2,"properties":{"mentions":"[]","cards":"[]","importance":"","subject":"","title":"","links":"[]","files":"[{"itemid":"ABB597A863CD8A2E!s5ce0be9da0eb48648f46c4920eeb1340","fileName":"requiem_draft.pdf","fileType":"pdf","fileInfo":{"itemId":"ABB597A863CD8A2E!s5ce0be9da0eb48648f46c4920eeb1340","fileUrl":"https://onedrive.live.com?cid=ABB597A863CD8A2E&id=ABB597A863CD8A2E!s5ce0be9da0eb48648f46c4920eeb1340","siteUrl":"","serverRelativeUrl":"","shareUrl":"https://1drv.ms/b/c/abb597a863cd8a2e/EZ2-4FzroGRIj0bEkg7rE0ABaiAlbUxZOZqmEBW970OlvA","shareId":"u!aHR0cHM6Ly8xZHJ2Lm1zL2IvYy9hYmI1OTdhODYzY2Q4YTJlL0VaMi00Rnpyb0dSSWowYkVrZzdyRTBBQmFpQUliVXhaT1pxbUVCVzk3ME9sdkE"},"fileChicletState":{"serviceName":"p2p","state":"active"},"@type":"http://schema.skype.com/File","version":2,"id":"ABB597A863CD8A2E!s5ce0be9da0eb48648f46c4920eeb1340","baseUrl":"","objectUrl":"https://onedrive.live.com?cid=ABB597A863CD8A2E&id=ABB597A863CD8A2E!s5ce0be9da0eb48648f46c4920eeb1340","type":"pdf","title":"requiem_draft.pdf","state":"active","chicletBreadcrumbs":null,"providerData":"","botFileProperties":{},"isUploadError":null,"progressComplete":null,"permissionScope":null,"filePreview":{"previewUrl":"","previewHeight":0,"previewWidth":0},"sharepointIds":null,"publication":null,"site":null}]","formatVariant":"TEAMS","languageStamp":"languages=en:100;nl:72;id:69;length:25;&detector=Bling"},"skypeGuid":"undefined","translation":"undefined","isSanitized":true,"isConversationLastMessage":false,"isConversationLastMessageSanitized":false,"originalNonLieMessage":"undefined","annotationsSummary":"undefined","hasAnnotated":"undefined","state":1,"dlpData":"undefined","inlineImages":"undefined","isSentByCurrentUser":false,"draftDetails":"undefined","pinnedTime":"undefined","crossPostData":"undefined","sendPipelineStatus":null,"streamingMetadata":"undefined","skypeeditedid":"undefined"}},"messageSearchKeys":[0,"19:uni01_vl6knbyum5ca36dmm36w2omnr6f42euhfe4jps4ihqcdu75r6vdq@thread.v2_1757589802321"],"lastUpdatedBy":2,"consumptionHorizon":"undefined","consumptionHorizonBookmark":"undefined","consumptionHorizonBookmarkVersion":"undefined","followState":"undefined","rcmlastUpdatedTime":"undefined","rcmSource":"undefined","isRootSaved":"undefined","savedL2Ids":"undefined"}

The attachments sent were linked. Under the entry `shareUrl`, the respective original attachment could be viewed and downloaded via the stored link without prior authentication:

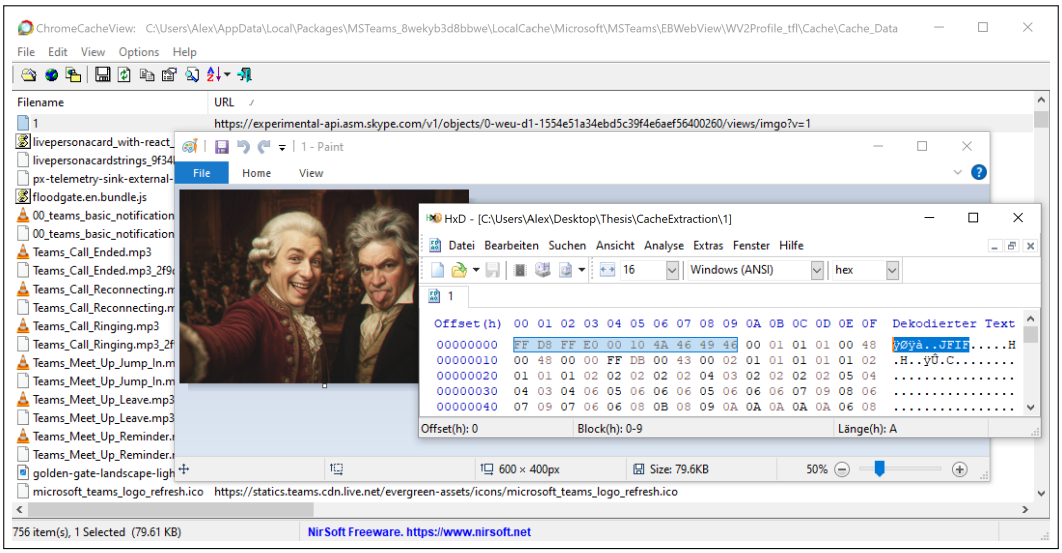



(a) Anhang 1: requiem_draft.pdf




(b) Anhang 2: selfie.jpg

Furthermore, the image displayed in the chat could be found in the Teams cache after prior processing by the tool ChromeCacheView[5] (v2.52):

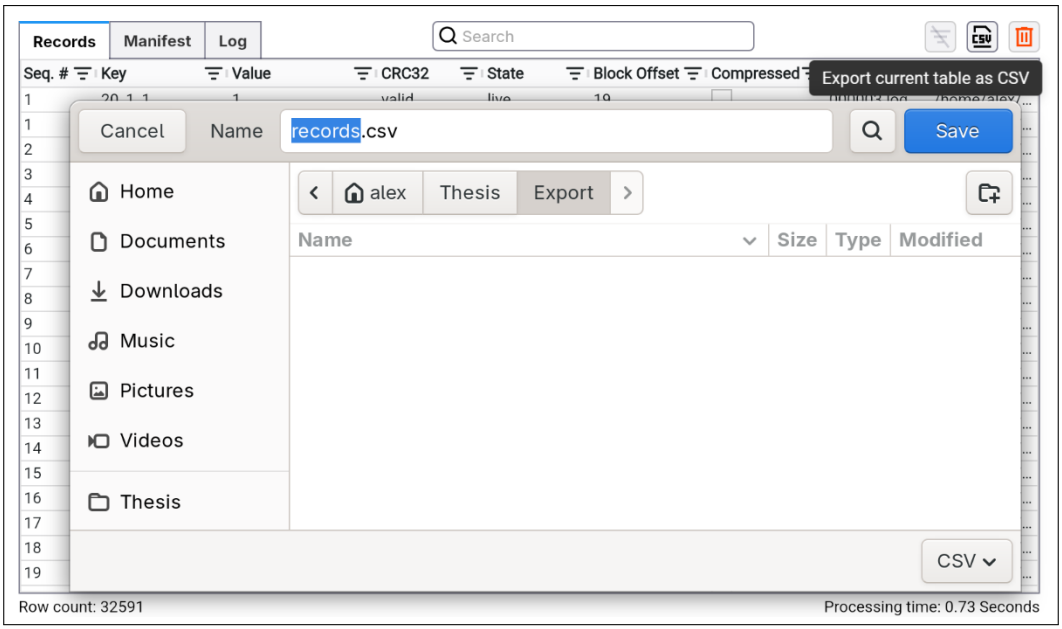



In this case, the image stored in the cache had the same hash as the original. Another test with a larger PNG image (3000×2000 px) showed that, in addition to a reduction in size, the image was also converted to JPG format for the cache and thus for the chat display.

[5] `https://www.nirsoft.net/utils/chrome_cache_view.html`

## 6.4 Data Extraction Script

Based on the findings, a script (Appendix: Data Extraction Script) was created to extract relevant data from the chats. To do this, all database entries were first exported as a CSV file using the parser:




After passing the CSV file (records.csv, 18.3 MB) to the script, all saved chats, including draft messages and associated attachments, were successfully extracted and displayed chronologically:

```
"Datetime(UTC)","Sender","Message","Attachments","ShareURL"
"2025-09-11 10:33:18","Wolfgang Mozart","Hey Johann, lol your fugues are too long 😂","",""
"2025-09-11 10:34:08","Johann Bach","At least mine don't sound like opera karaoke","",""
"2025-09-11 10:34:33","Wolfgang Mozart","Fair enough. Been experimenting with pianoforte, much more dynamic than the harpsichord.","",""
"2025-09-11 10:34:57","Johann Bach","Interesting. But does it handle counterpoint as gracefully?","",""
"2025-09-11 10:35:26","Wolfgang Mozart","Counterpoint? Pfft, I'm more about melody and drama 🎭","",""
"2025-09-11 10:36:14","Johann Bach","Without structure, drama collapses. Try a fugue sometime, Wolfgang 😉","",""
"2025-09-11 11:23:22","Wolfgang Mozart","Btw, here is the draft...","requiem_draft.pdf","https://1drv.ms/b/c/abb597a863cd8a2e/EZ2-4FzroGRIj0bEkg
"2025-09-11 11:23:48","Johann Bach","Half the notes are missing... again","",""
"2025-09-11 11:24:14","Wolfgang Mozart","Lol I'll finish it later, I'm busy with a party 🎉","",""
"2025-09-11 11:25:02","Johann Bach","While you drink, I'll write another cantata. Let's see whom history remembers.","",""
"2025-09-11 11:26:41","Wolfgang Mozart","Ha! Let's just hope the critics don't tag it as ""too complex"" again 😂","",""
"2025-09-11 11:26:52","Johann Bach","😠","",""
"2025-09-11 11:39:51","Wolfgang Mozart","","selfie.jpg","https://1drv.ms/i/c/abb597a863cd8a2e/EdGXc5wNdzlJvqH1vassy90BDoZBDtaD8CBEEyY3EFuA8A"
"2025-09-11 11:41:01","Johann Bach","Greetings to Ludwig! Hopefully the music wasn't too loud, as we all know, that's hard on the ears...","",""
"2025-09-11 11:54:06","Alexander Hübert","Hello Johann, I'm testing Teams for my thesis. Can you send me a short reply please?","",""
"2025-09-11 11:54:48","Johann Bach","Of course! Even a fugue can be concise if necessary 😉","",""
"2025-09-11 11:57:51","[DRAFT]","This is another drafted message for Wolfgang...","",""
"2025-09-11 11:58:32","[DRAFT]","This drafted message with attachments is never sent...","squirrel.jpg, requiem_draft.pdf, selfie.png",""
```

In the same way as described above, messages from the chat partners could also be successfully extracted from the associated databases of the Vivaldi web browser.

# Chapter 7

# Conclusion and Outlook

Through in-depth code and runtime analysis of LevelDB, the structure and layout of the database could be explored and demonstrated in detail. In the subsequent excursus, the use of the database in common and practical scenarios was illustrated by describing the use of LevelDB in Chromium-based applications, in particular via the use of the Web APIs *Web Storage* and *IndexedDB*.

Based on these findings, a requirements analysis and design concept were followed by the development of the forensic parser for LevelDB, which can display all available database information and is intended to facilitate forensic work with this database. Two applications were implemented based on the developed parser library, which complement each other in their functionalities. Among other things, the GUI parser enables clear display and quick searching of the textual content of several databases simultaneously, while the CLI parser allows detailed analysis of the contents of individual database files, including all available metadata. The two tools are thus designed to provide both an efficient overview and in-depth forensic investigation of LevelDB databases.

The following validation of the parser was successful. This involved not only selecting self-generated data for the tests, but also using real data that had been collected from various sources over several years of use and included more than half a million database entries.

Nevertheless, the validation also revealed opportunities for optimization and extension of the parser that was created. For example, when analyzing numerous large databases, the GUI parser consumes a large amount of RAM, as all processed data is currently stored in the main memory. This could be solved by offloading and streaming the data from mass storage, although this would result in losses in processing time and search performance.

When working with data stored via web APIs, for example, it is also desirable to switch the encoding from UTF-8 to UTF-16 in order to be able to display and export special characters and emojis correctly. However, during the excursion, it became apparent that the web APIs sometimes use the Latin-1, UTF-16LE, and UTF-16BE encodings simultaneously, thus mixing them. The most technically precise, but also the most complex approach is to develop a decoder/deserializer for the respective web API in order to extend the existing LevelDB parser on this basis. Given the widespread use of LevelDB in connection with data stored via web APIs, the future development of such extensions is a high priority.

Furthermore, a hex dump display is planned for the GUI parser, which should make it possible to not only evaluate textual content, but also to view complete entries at the byte level in hexadecimal representation. In the future, the GUI parser could also be expanded to display complete metadata, similar to what is possible with the CLI parser.

As part of the application example, the effective use of the parser with real data from the Microsoft Teams application was finally demonstrated. All that was required was to pass the main directory to the GUI parser in order to process all 31 LevelDB databases contained therein simultaneously and then analyze them. This enabled the efficient and complete assignment of relevant entries to the databases and a targeted structural analysis of the chat entries. Based on the findings obtained in this way, an external script was created to extract relevant content from the exported CSV data. The relevant data of the chat partners could also be exported and extracted, which used Teams via a Chromium-based web browser. Here, it would also be worth considering to extend the parser with a built-in script engine in order to be able to extract the desired data internally without having to resort to external scripts.

# References


[1] Lucia-Raisa Verstein and Alexander Hübert. *Verhaltens- und Netzwerkanalyse der Android-Anwendung „Notion“*. Tech. rep. Hochsch. Albstadt-Sigmaringen, 2024.

[2] Google. *LevelDB Source Code*. URL: `https://github.com/google/leveldb` (visited on 01/18/2025).

[3] Alexander Bilz. “Digital Forensic Acquisition and Analysis of Artefacts Generated by Microsoft Teams”. Master’s thesis. Dundee: Abertay University, 2021.

[4] Web Hypertext Application Technology Working Group (WHATWG). *Web Storage Specification*. URL: `https://html.spec.whatwg.org/multipage/webstorage.html` (visited on 04/28/2025).

[5] World Wide Web Consortium (W3C). *Indexed Database API 2.0*. URL: `https://www.w3.org/TR/IndexedDB-2` (visited on 04/28/2025).

[6] Christian Hummert and Dirk Pawlaszczyk, eds. *Mobile Forensics – The File Format Handbook*. 1st ed. Cham, Switzerland: Springer, 2022. ISBN: 978-3-030-98467-0.

[7] Chad Tilbury. *Google Chrome Platform Notification Analysis*. URL: `https://www.sans.org/blog/google-chrome-platform-notification-analysis` (visited on 04/29/2025).

[8] Alex Caithness. *Hang on! That’s not SQLite! Chrome, Electron and LevelDB*. URL: `https://www.cclsolutionsgroup.com/post/hang-on-thats-not-sqlite-chrome-electron-and-leveldb` (visited on 01/18/2025).

[9] CCL Solutions Group. *Chromium Reader Source Code*. URL: `https://github.com/cclgroupltd/ccl_chromium_reader` (visited on 01/18/2025).

[10] Google. *LevelDB Implementation*. URL: `https://github.com/google/leveldb/blob/main/doc/impl.md` (visited on 05/19/2025).

[11] William Pugh. “Skip lists: a probabilistic alternative to balanced trees”. In: *Communications of the ACM* 33.6 (1990). URL: https://dl.acm.org/doi/abs/10.1145/78973.78977 (visited on 05/16/2025).

[12] Google. *Protocol Buffers Documentation*. URL: https://protobuf.dev (visited on 05/01/2025).

[13] Google. *LevelDB Table Format*. URL: https://github.com/google/leveldb/blob/main/doc/table_format.md (visited on 06/05/2025).

[14] Mozilla. *IndexedDB key characteristics and basic terminology*. URL: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Basic_Terminology (visited on 05/05/2025).

[15] Mozilla. *Storage quotas and eviction criteria*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Storage_API/Storage_quotas_and_eviction_criteria (visited on 04/29/2025).

[16] Alex Caithness. *Chromium Session Storage and Local Storage*. URL: https://www.cclsolutionsgroup.com/post/chromium-session-storage-and-local-storage (visited on 04/30/2025).

[17] The Chromium Project. *IndexedDB Documentation*. URL: https://github.com/chromium/chromium/tree/master/content/browser/indexed_db/docs (visited on 05/05/2025).

[18] Alex Caithness. *IndexedDB on Chromium*. URL: https://www.cclsolutionsgroup.com/post/indexeddb-on-chromium (visited on 05/02/2025).

[19] Byeongchan Jeong, Sangjin Lee, and Jungheum Park. “MIC: Memory analysis of IndexedDB data on Chromium-based applications”. In: *Forensic Science International: Digital Investigation* 50 (2024). URL: https://www.sciencedirect.com/science/article/pii/S2666281724001331 (visited on 05/08/2025).

[20] Alexander Hübert. *LevelDB Parser Source Code*. URL: https://github.com/huebicode/leveldb-parser.

# Appendix: Data Extraction Script

```python
#!/usr/bin/env python3
import sys
import re
from datetime import datetime, timezone

def extract_re_group(pattern, line, flags=0):
    match = re.search(pattern, line, flags)
    return match.group(1) if match else ''

def extract_chat(filename):
    with open(filename, 'r', encoding='utf-8') as file:
        extracted = []
        for line in file:
            message = extract_re_group(r'<p>(.*?)</p>', line)
            id_match = extract_re_group(r'"id"":""(\d{13})"', line)
            name_match = extract_re_group(r'"imDisplayName"":""([^"]+)"', line, re.IGNORECASE)
            attachment = re.findall(r'"fileName"":""([^"]+)"', line)
            attachment = ', '.join(attachment)
            share_url = re.findall(r'"shareUrl"":""([^"]+)"', line)
            share_url = ', '.join(share_url)

            draft_creation = extract_re_group(r'"draftCreationTime"":""(\d{13})"', line)
            draft_attachment = re.findall(r'"title"":""([^"]+)"', line)
            draft_attachment = ', '.join(draft_attachment)

            if ((id_match and name_match and not (message == '' and attachment == ''))
                or (draft_creation and not (message == '' and draft_attachment == ''))):
                if draft_creation:
                    extracted.append((draft_creation, '[DRAFT]', message, draft_attachment, ''))
                else:
                    extracted.append((id_match, name_match, message, attachment, share_url))

    # combine to remove duplicates
    combined = {}
    for timestamp, sender, msg, attachment, share_url in extracted:
        key = (timestamp, sender, attachment, share_url)
        # if not seen -> add OR identical fields, but message is not blank -> replace
        if key not in combined or (msg and not combined[key][2]):
            combined[key] = (timestamp, sender, msg, attachment, share_url)

    # header
    print("\"Datetime(UTC)\",\"Sender\",\"Message\",\"Attachments\",\"ShareURL\"")

    # output extracted content sorted by timestamp
    for timestamp, sender, msg, attachment, share_url in sorted(combined.values(), key=lambda x: int(x[0])):
        dt = datetime.fromtimestamp(int(timestamp) / 1000, tz=timezone.utc)
        print(f"\"{dt.strftime('%Y-%m-%d %H:%M:%S')}\",\"{sender}\",\"{msg}\",\"{attachment}\",\"{share_url}\"")

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print(f"Usage: {sys.argv[0]} <records.csv>")
        sys.exit(1)

    filename = sys.argv[1]
    extract_chat(filename)
```

# Declaration of Authorship

I hereby declare that I have written this thesis independently, using no sources or aids other than those indicated, and that all passages or ideas taken directly or indirectly from other sources are properly cited as such.

This thesis has not been submitted to any other examination board in this or a similar form and has not been published previously.

I hereby agree that this thesis may be checked for plagiarism in electronic form using appropriate software by the examiner.

Alexander Hübert
Düsseldorf, October 9$^{th}$, 2025