Technical Paper

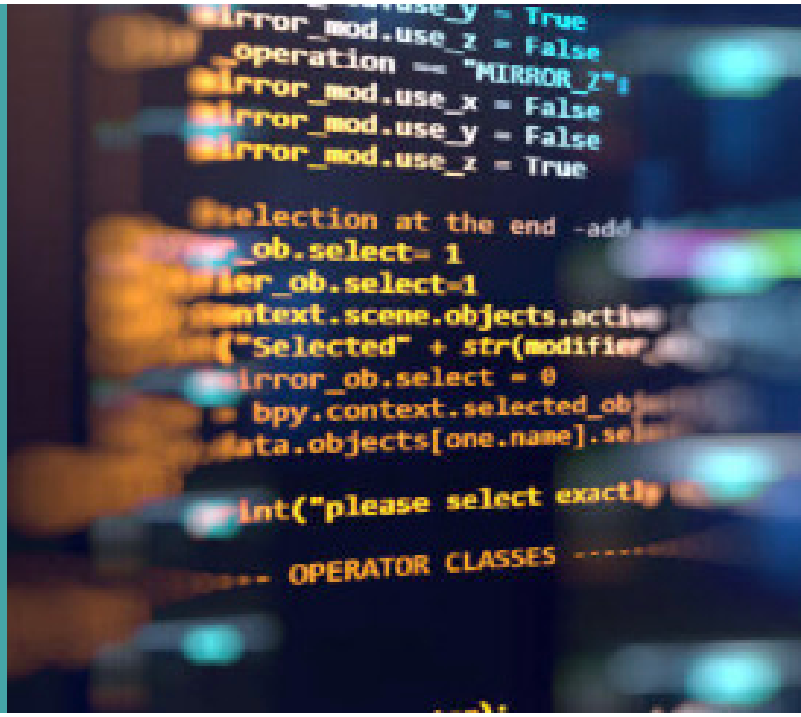
CCL SOLUTIONS GROUP

incorporating evidencetalks

Hang on! That's not SQLite!

Chrome, Electron and LevelDB

CCL's Principal Analyst Alex Caithness asks the question: After SQLite, what comes next? A must-read primer on LevelDB - tomorrow's ubiquitous format?



SQLite has become a ubiquitous data storage format for digital forensic practitioners to consider. First popularised by smartphone platforms it now forms part of almost every investigation in one form or another.

SQLite's ubiquity was built upon the growing market share of the platforms that used it extensively (we're talking iOS and Android – name a more iconic duo, I'll wait), so it's interesting to ask the question: what's the next platform, and what's the next data format?

CCL SOLUTIONS GROUP incorporating evidencetalks

Browser as OS, Webpage as App

Webpages can do very complex things. The trio of HTML5, CSS and JavaScript (and latterly, increasingly WebAssembly, aka WASM), plus the optional extra of offloading processing to the cloud, enable coders to make webpages which fulfil the role of a traditional application – a "Web App".

For example, I'm drafting this blog in a version of Microsoft's Word for Office 365 running in the browser – indeed the entire Office 365 suite is available running in the browser (other web-based office suites are available of course, such as Google's G Suite). We also have web-based image editing apps like Photopea and video editors such as Adobe Spark. DOSBox has been ported to JavaScript meaning that the browser can natively run software written for DOS which means, yes – the browser can run DOOM.

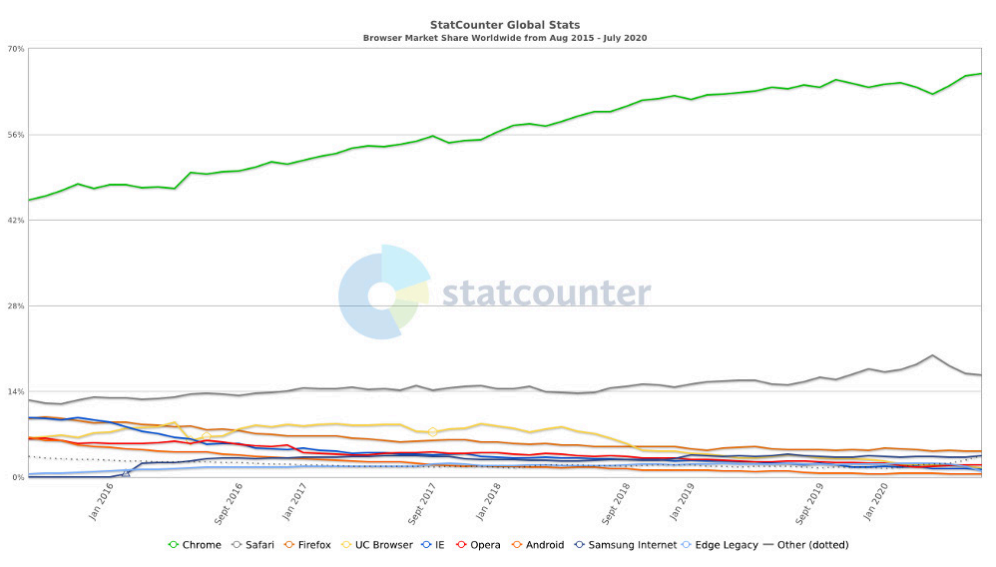
Although web browsers provide APIs (Application Programming Interface) for opening and saving data to a user's local machine, making use of them is often clunky. On one side, making use of the user's host system requires an understanding of the operating system currently in use, and on the other the browser tries to "sandbox" Web Apps, separating them from each other, and restricting their access to the host system for security reasons.

Because of this, data persistence is often achieved by some combination of storing data in the cloud and making use of a variety of browser managed storage mechanisms; keep this last part in mind as you read on – we'll be returning to this idea.

CCL SOLUTIONS GROUP
incorporating evidencetalks

The Browser Wars

Spoiler: Chrome won.



Safari and Chrome actually have a shared lineage in their rendering Engine – Safari using WebKit and Chrome using Blink, which is based on WebKit. From that we can see that around 82% of browsers currently in use are based on the WebKit rendering engine, which is probably a good thing for standardisation, but not so great for competition.

But it actually goes further (or 'gets worse', depending on your how you look at things) – on the chart above alone, Opera, Android Browser and Samsung Internet are all based on the Chromium codebase (Chromium being the open source version of Chrome) and the new version of Edge that now ships with Windows 10 is also based on Chromium. As these browsers are all sharing aspects of a single codebase many of the artefacts associated with these browsers will also be shared – and convergence is usually good news from a digital forensics standpoint.

Because of this convergence I've started referring to these browsers that borrow heavily from Chrome (or Chromium) as being "Chrome-esque". Chrome is winning a war on another front as well – but we'll get to that in a bit.

CCL SOLUTIONS GROUP incorporating evidencetalks

IndexedDB

As Web Apps become more and more common, the need to persist data between sessions is often necessary to properly provide the functionality on offer. A word processor running the browser is all well and good, but if you can't save your work as you go, it becomes a risky business.

Of course, storing data in the cloud has become standard practice – being able to access your data from any location and on any device is an expectation of most users.

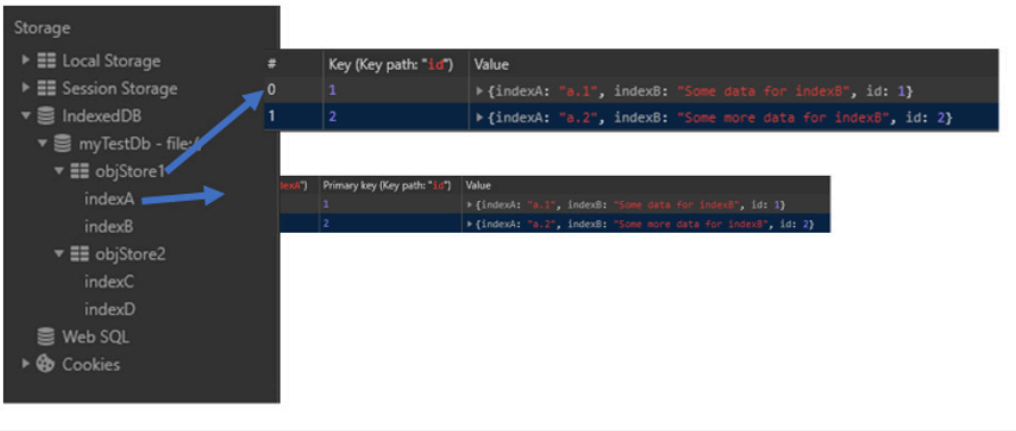
That being said, web apps should also operate offline in situations where that functionality makes sense – as would be the case for a word processor for example.

For this, data needs to be stored locally, if only ephemerally. Local data storage can also be used for speed, where resources need to be accessed multiple times, for tracking, to provide functionality for browser extensions, and so on.

As mentioned, browsers tend to abstract websites away from the underlying file system and instead provide APIs that provided storage to websites via an API. One such API is IndexedDB.

IndexedDB provides web developers a means to store JavaScript objects across multiple datastores on a per-domain basis. The data types allowable in the store comprise all the primitive types such as numbers, Booleans, strings, binary blobs and more, as well as the collection types that allow the arrangement of that data into a hierarchical structure if needed. The data can also be addressed via multiple indexes based upon properties of the object if needed.

To create an analogy with "classic" relational databases, you might consider each domain to have a single database which contains multiple "object stores" (tables), which contain a sequence of objects (records or rows). Records can be addressed based on their primary key, or any index field set on the object store.



For a concrete example of IndexedDB being used in the wild, try creating a document in Google Docs and then setting it to be "Available offline"; the data is stored by IndexedDB in the "Documents" object store, rather than on the local filesystem.

Compared to other storage APIs, IndexedDB shines in terms of its flexibility – good old Cookies provide only a very small storage space per record and needs to be text based; Web Storage (Local Storage and Session Storage) is again text-based only; Web SQL and Application Cache (aka AppCache) are both deprecated legacy technologies (and never formally standardised in the case of Web SQL).

Finally there is the "FileSystem API" which is certainly interesting as it allows a Web App to make use of a file-system-like structure to store data locally; it is currently still officially an experimental API, although it is well supported by mainstream browsers. The FileSystem API on Chrome also makes use of the same technology as IndexedDB as it happens, albeit storing data in a different structure – but that's for another blog.

The thing to keep in mind about IndexedDB is that it is an API: a way for developers to interact with the browser in order to store and retrieve data – how the browser chooses to store the data is an implementation detail that is up to each browser (it's just odds on that the browser will be Chrome, or Chrome-esque).

Electron

Before we jump into the bits and bytes of how IndexedDB is stored by our Chrome-esque browsers, let's talk about Web Apps again for a moment.

We've established that modern web development allows one to create incredibly rich app experiences, but the desktop still needs apps, and they're going to be running natively on the operating system, right? Right?

The thing is: web development technologies have matured so far now that in many cases it can be desirable to create a Web App because you can deploy that anywhere that can run a browser. But what if you want your app to look and act like a "real app"?

That's where Electron (formally Atom Shell) comes in. Electron is an open source framework maintained by GitHub for building cross-platform applications based on web technologies. Electron allows a developer to bundle up the front-end of a website or web app – built with HTML5, CSS and JavaScript – with a back-end webserver that runs locally on a user's machine. The back-end capabilities are provided by a technology called Node.js, which also enriches the Web technologies by providing additional methods for interacting with the host OS. The rendering of the frontend? That's provided by Chromium. An Electron application window looks like a native application – but it's really a browser with all the web navigation stuff stripped out.

So how common are these applications? Well if you're running an up-to-date version of Windows, you almost certainly have at least one Electron based application already installed: Skype. Yep: Skype is based on Electron which means that it's Chromesque and at the time of writing much of the chat information is persisted through IndexedDB.

CCL SOLUTIONS GROUP incorporating evidencetalks

Name	Size	Type	Date Modified
blob_storage	1	Directory	2020-02-24 15:49:46
Cache	1	Directory	2020-02-24 15:50:28
Code Cache	1	Directory	2020-02-24 15:49:45
CS_skylib	1	Directory	2020-02-24 15:55:09
databases	1	Directory	2020-02-24 15:50:08
GPUCache	1	Directory	2020-02-24 15:49:46
IndexedDB	1	Directory	2020-02-24 15:50:07
Local Storage	1	Directory	2020-02-24 15:49:46
logs	1	Directory	2020-02-24 15:49:45
media-stack	1	Directory	2020-02-24 15:50:10
skylib	1	Directory	2020-02-24 15:50:09
SkypeRT	1	Directory	2020-02-24 15:50:10
$I30	8	NTFS Index All...	2020-02-24 16:02:21
Cookies	20	Regular File	2020-02-24 16:02:41
Cookies-journal	0	Regular File	2020-02-24 16:02:41
Cookies.FileSlack	12	File Slack	
device-info.json	1	Regular File	2020-02-24 15:49:45
ecscache.json	1	Regular File	2020-02-24 15:49:46
en-US-8-0.bdic	437	Regular File	2020-02-24 15:50:28
lockfile	0	Regular File	2020-02-24 15:49:45
Network Persistent State		$I30 INDX Entry	
NETWOR~1		$I30 INDX Entry	
Preferences	1	Regular File	2020-02-24 15:50:38
QuotaManager	52	Regular File	2020-02-24 16:02:41
QuotaManager-journal	0	Regular File	2020-02-24 16:02:41
QuotaManager.FileSlack	12	File Slack	
settings.json	1	Regular File	2020-02-24 15:51:20
TransportSecurity		$I30 INDX Entry	
TRANSP~1		$I30 INDX Entry	

At a glance you'd be forgiven for thinking that this directory listing came from Chrome, but no: this is Skype

But a quick survey of apps we've investigated we've seen the following:

- Discord
- GitHub Desktop
- Signal (desktop)
- Skype
- Slack
- Microsoft Teams
- WebTorrent (desktop)
- WhatsApp (desktop)
- Microsoft Yammer

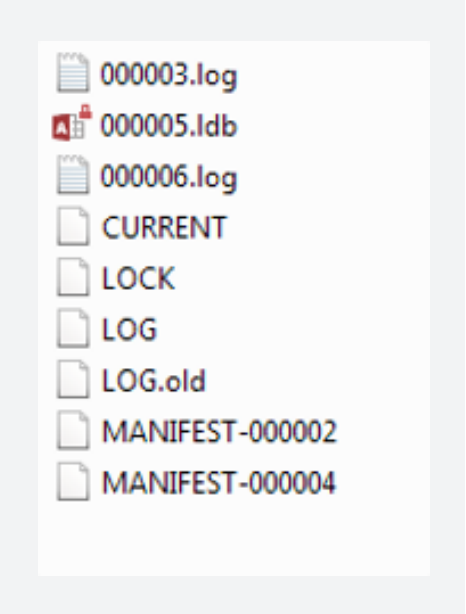
All built on top of Electron; all Chrome-esque.

CCL SOLUTIONS GROUP incorporating evidencetalks

LevelDB Overview

So, what format do the Chrome-esque browsers (and other apps) use to actually store the IndexedDB data? Is it SQLite like seemingly everything else? Might they be kicking things oldskool with a selection of XML files perhaps? It's JavaScript objects that are being stored, so maybe JSON would work?

The answer in this case is LevelDB. And if you've ever seen a folder like the one below, you've seen LevelDB:



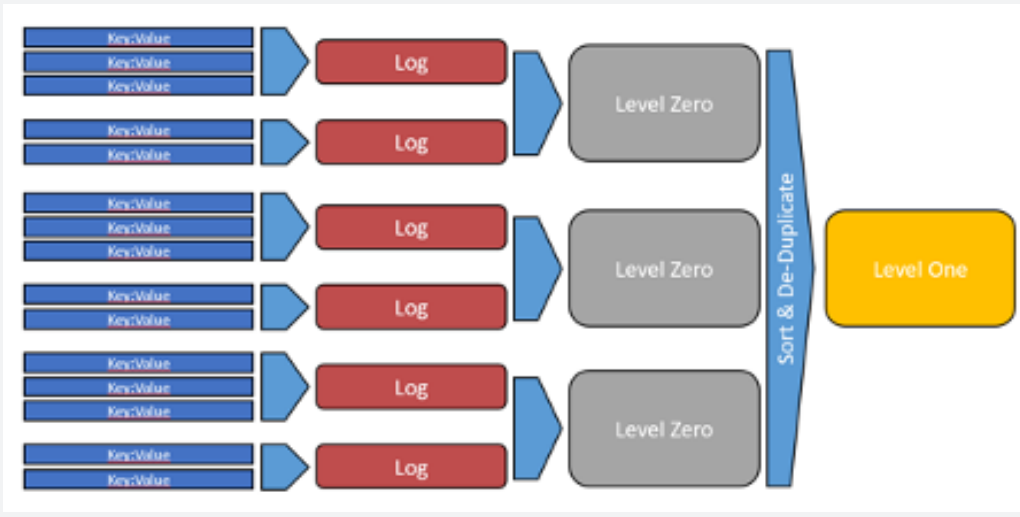
LevelDB is an on-disk key-value store where the keys and values are both arbitrary blobs of data. Each LevelDB database occupies a folder on the file system.

The folder will contain some combination of files named "CURRENT", "LOCK", "LOG", "LOG.old" and files named "MANIFEST-######", "######.log" and "######.ldb" where ###### is a hexadecimal number showing the sequence of file creation (higher values are more recent). The ".log" and ".ldb" files contain the actual record data; the other files contain metadata to assist in reading the data in an efficient manner.

When data is initially written to a LevelDB database it will be added to a ".log" file. "Writing data" to LevelDB could be adding a key to the database, changing the value associated with a key or, indeed, deleting a key from the database. Each of these actions will be written to the log as a new entry, so it is quite possible to have multiple entries in the log (and, indeed, ".ldb" files) relating to the same key. Each entry written to LevelDB is given a sequence number, which means that it is possible to track the order of changes to a key and recover keys that have been deleted (indeed, it is actually more effort to exclude old and deleted entries when reading the database!).

When a log file reaches a particular size (4 MB by default), the data it holds will be converted into a permanent table file (a ".lbd" file) and a new log file started. The ldb files that are created by converting a log file are said to be "level 0" files and will contain the same data from the entries found in the log file - including all the updated or deleted records.

As the number of level 0 files reaches a threshold (4 files by default), a "compaction" will be performed where the records from the level 0 files will be sorted and deduplicated based on their keys and moved into a new "level 1" file. These compactions continue in the same vein through the lifetime of the database, adding new levels as records from earlier levels are compacted, further sorting the data and removing redundant data. The deduplication of keys means that records in level 1 files and up no longer contain the old or deleted versions of records (although, there may still be newer versions of those records available in earlier levels or in the current logs).



CCL SOLUTIONS GROUP incorporating evidencetalks

LevelDB Log File Format

The LevelDB ".log" files are broken up into blocks of a fixed size (32 kB). The blocks contain a block header followed by data related to log entries. The header is 7 bytes long and has the following format:

Offset	Length	Meaning
0	4	CRC-32 of block data
4	2	Int16 data length for this block
6	1	Block type (see below): 1: Full 2: First 3: Middle 4: Last

As data can be added to LevelDB in batches, more than one block in the log file may be needed to contain all records for that batch. In this case a number of blocks can be designated as forming part of a batch: the first block being marked as the "Start" of the batch; the final block being marked as the "Last"; and all intermediate blocks marked as being "Middle" blocks. If all records from a batch fit on a single block it will be marked as being a "Full" block (multiple batches could be stored in a single Full block if they fit).

The data is written to the log in the "batch format" which begins with a 12 byte batch header:

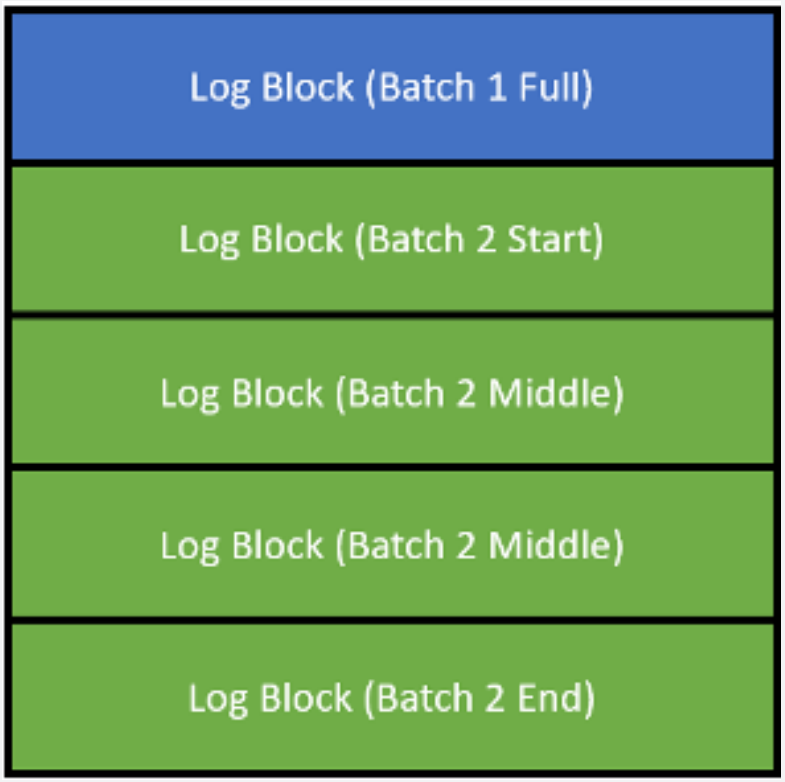
Offset	Length	Meaning
0	8	Int64 sequence number
8	4	Int16 data record counts

This is followed by a number of record format entries as specified in the batch header:

Offset	Length	Meaning
0	1	Record state (0 + deleted; 1 = live)
1	1-4	VarInt32 LE key_length
...	key_length	Key (blob)
...	1-4	VarInt32 LE value_length
...	value_length	Value (blob)

The sequence number of the first record will be the value given in the batch header, all subsequent sequence numbers can be inferred by incrementing that value for each record. The record format makes use of VarInts (variable length integers) for encoding numeric values; more details can be found here:

https://developers.google.com/protocol-buffers/docs/encoding#varints



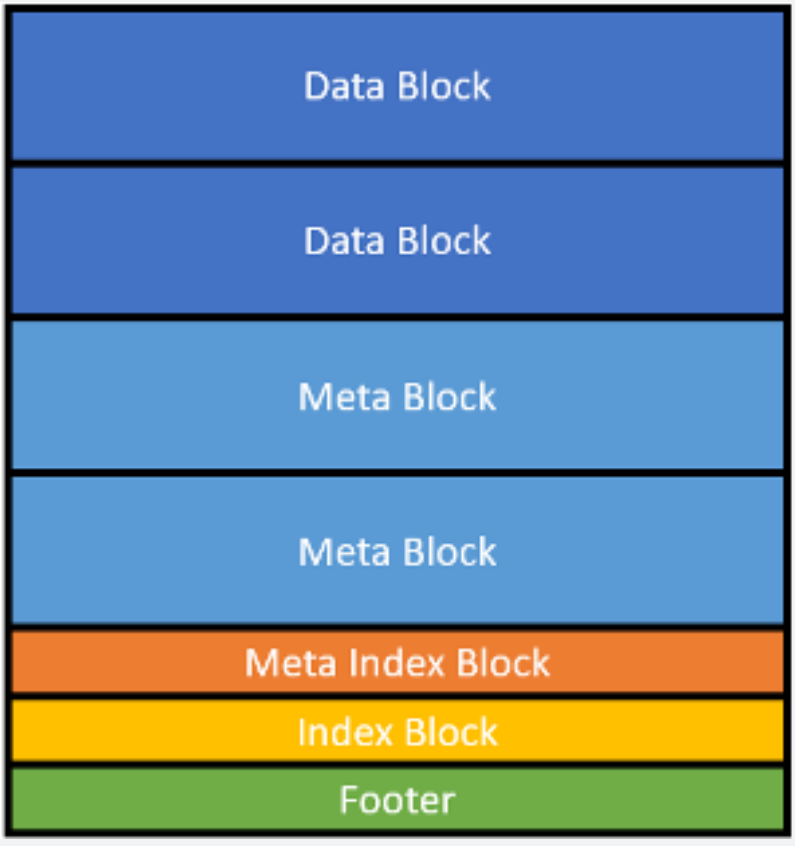
From a data recovery perspective, the log file format offers some great benefits: keys and values are stored in full, without any compression applied, therefore string searches and carving will be successful in a lot of cases. It is worth considering that as batched data can flow across block boundaries it will be interrupted by the 7 byte block header if it spans multiple blocks. If searching and carving is the primary concern then, it is worth pre-processing the log by removing 7 bytes every 32 kB to optimise the chances to recover data.

CCL SOLUTIONS GROUP incorporating evidencetalks

LevelDB ldb File Format

The ldb file format it a little more involved than its log file counterpart. The ldb files are made up of Blocks (slightly confusingly this term is reused in the Log, but the structure and contents are different). Unlike the blocks in the log files, blocks in ldb files do not have a fixed length.

All of the blocks in the ldb files are structured the same way, but do different jobs: data is held in Data blocks, metadata is stored in Meta blocks, the locations and sizes of Meta blocks are stored in the Meta Index block, the locations and sizes of Data blocks are stored in the Index block.



The footer of the ldb file is a special case: it is 48 bytes in length and is always found at the end of the file. It begins with the locations and sizes of the Index block and Meta Index block; the final 8 bytes contain the file signature: 0x 57 FB 80 8B 24 75 47 DB.

Whenever I've mentioned the "location and size" of a thing in the ldb file, this information is stored as what the developers of LevelDB describe as a BlockHandle structure. A BlockHandle is made up of two little endian VarInts, the first being the absolute offset, the second being the size in bytes.

Blocks will always contain zero or more BlockEntry structures followed by a "restart array" (the restart array is useful for reading the data if you need to do it fast by skipping keys, but it isn't actually essential for reading the data sequentially). Blocks will always be followed by a 5 byte long "trailer" structure; the length of the trailer is not included in the length of a block as defined by a BlockHandle. The block trailer is made up of the following values:

Offset	Length	Meaning
0	1	Compression type (0 = None; 1 = Snappy)
1	4	CRC-32 of the block data

You might have noticed that the table above mentions compression. Data in ldb files can be compressed using the Snappy compression algorithm (https://github.com/google/snappy) designed by Google. Snappy is a compression algorithm where speed is always preferred over high compression ratios. Because of the way that it operates, it is normal to see data appearing to be more or less uncompressed and legible towards the start of a block but becomes more and more "broken" looking as repeated patterns are back-referenced. It is important to understand that this compression is in use in the long-term storage for LevelDB records: naïve searching and carving is unlikely to be successful when applied to this data, without first decompressing the contents of the blocks.

The BlockEntry structures in the Data blocks contain the keys and values for the records. The format of the BlockEntry is:

Offset	Length	Meaning
0	1-9	VarInt shared_key_length
...	1-9	VarInt inline_key_length
...	Inline_key_length	Inline portion of key (blob)
...	1-9	VarInt value_length
...	value_length	Value (blob)

In the previous table there are two different lengths for the key – the shared key and the inline key. The keys in a single block are encoded using key sharing. Key sharing means that when a key shares a common prefix with the previous key (which is not unlikely as the keys should be ordered as far as possible), rather than storing the shared start of the key, only the part at the end that is different will actually be stored inline, and the shared prefix can be inferred from the previous entry, eg:

Key	Shared Length	Inline Length	Inline Key Content
thisisKeyOne	0	12	thisiskeyOne
thisisKeyTwo	9	3	Two
thisisKeyThree	10	4	hree
theForthKey	2	10	eForthKey

As with the compression, it is important to note that key sharing is used in the ldb files. As before, if string searching is being used to identify data, if that data tends to reside near the start of a key, for records in close proximity this data may be "shared out" and not be captured until the keys have been expanded.

In the final 8 bytes of every key in the ldb file there is additional metadata (this is not part of the key as a database user would see it). The metadata contains a 56-bit sequence number in the 7 most significant bytes and a state in the least significant byte, which is the state of the record where zero is deleted and one is live.

CCL SOLUTIONS GROUP incorporating evidencetalks

Next Steps

In a future blog we will take a look at the specific format used by Chrome's IndexedDB implementation and work through how to go from LevelDB back to a representation of the JavaScript objects stored. In the meantime, I hope this blog has helped shed some light on LevelDB and drawn attention to some of the potential opportunities and pitfalls that LevelDB presents from a data recovery and forensics perspective.