Chapter 5
SQLite

Dirk Pawlaszczyk

Abstract SQLite is, without doubt, the most widely used database system worldwide at the moment. The single file database system is used, among other things, in operating systems for cell phones, such as Android, iOS or Symbian OS. On a typical smartphone, we usually find several hundred SQLite databases used by a wide variety of apps. Due to its widespread use, the database format is of particular importance in mobile forensics. It is not uncommon for the suspect to try to cover his tracks by deleting database content. Recovering deleted records from a database presents a special challenge. In this chapter, the on-disk database format of the SQLite database system is highlighted. Therefore, we take a closer look at the database header as well as record structure on a binary level. We first examine the structure of the data. Recovery options for erased records are discussed as well. Special attention is paid to the slack areas within the database: unallocated space, Freelist as well as free blocks. In this context, we discuss basic techniques for carving and acquisition of deleted data artefacts. Despite the main database format and recovery options, temporary file types like write-ahead logs and rollback journals are analyzed as well.

5.1 Introduction

A large amount of data is being stored and processed in relational databases. The most widely used database system in the world is undoubtedly SQLite since it is the default solution for the Android and iOS operating systems. So it is not surprising, that web browsers, messenger services and mobile applications employ the free and serverless database solution as their storage format of choice [61],[60]. At the moment, there are more than a trillion SQLite instances in active use [81]. In the vast majority of criminal investigations involving information technology, one task is to make information stored in such databases accessible. Evidence acquisition for

University of Applied Sciences (Hochschule Mittweida), Technikumplatz 17, 09648 Mittweida, Germany, e-mail: pawlaszc@hs-mittweida.de

C. Hummert, D. Pawlaszczyk (eds.), *Mobile Forensics – The File Format Handbook*,
https://doi.org/10.1007/978-3-030-98467-0_5

databases is traditionally made with SQL, a powerful query language. Also, SQLite supports most of the SQL language commands. In this way, the data can be accessed with one of the freely available viewers. Unfortunately, this form of analysis usually does not allow access to deleted records or temporary data content such as recently added but not committed entries. This creates the need for alternative ways to analyze such databases forensically.

5.2 The SQLite File Structure

SQLite is a single-file database engine, i.e., all tables are managed in only one file on disk. There is no intermediary server process; an application has to communicate with first, for storing data. It does not work this way. Instead, the database can be integrated directly into an application. Therefore, it provides a library and an easy to use programming interface. This fact has significantly contributed to the current spread and popularity of the program. We will discuss the basic structure of a database before turning to the details of carving for data records.

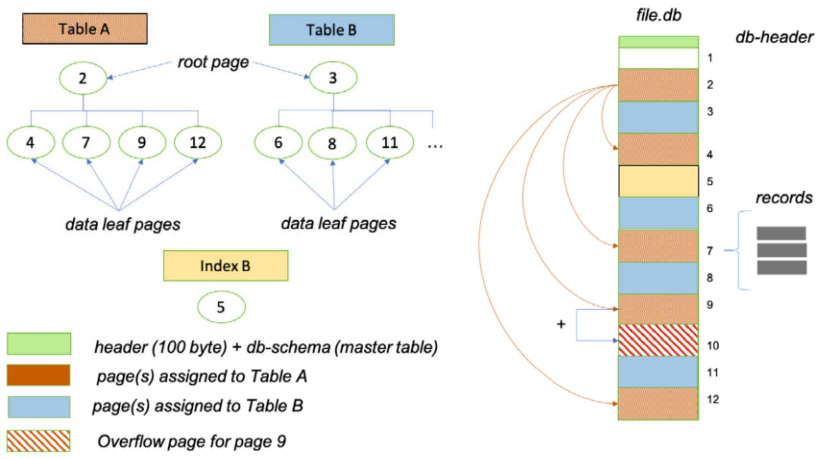


Fig. 5.1: Schematic structure of a SQLite database

Like most structured binary formats, the database file starts with a header part [80]. Its size is exactly 100 bytes. Beyond this, the database file is divided into pages of equal size. The file size is thus always a multiple of the page size. A page number uniquely identifies a single page, whereas the first page has the number one. The default page size usually is 4096 bytes. However, it can be adjusted if necessary to a minimum of 512 bytes and a maximum of 64KB [32]. Of course, the header is part of the first page. In a relational database system, all data is stored in tables. This is also the case with SQLite. In turn, a table is distributed over one or more pages of the database on the binary level (see Fig. 5.1). Each data page again contains one or more records, for precisely one table. To access and acquire all records of a particular table, we must first determine which pages of the database are associated with this table. This information can again be taken from the first page of the database. Besides the header string, this page contains one more piece of information - the database schema. Necessary information such as the root page numbers, column names, and column types of the tables are stored here, in a data structure called *SQLite_Master Table*. We will discuss the details of this table in sect. 5.2.3. To represent a table and its pages, SQLite uses a balanced tree data structure (B+tree) under the hood. In a B+tree, the raw data elements are stored exclusively in the leaf nodes, while the inner nodes contain only links. Since the maximum size of a page is limited from above, we can gain more space for links or branches in the inner nodes by moving the leaves' data records. Moreover, this limits the height of the tree. Since data elements are normally accessed via the tree's root, a lower height reduces the number of nodes to be traversed. Many relational database systems manage their records in this way.

Table 5.1: SQLite page types and byte flags

Page Type	1st Byte in Page
table b-tree interior page	0x05
table b-tree leaf page	0x0d
index b-tree interior page	0x02
index b-tree leaf page	0x0a
overflow page	0x00 (for db-size < 64GB)
freelist page	0x00 (first 8 bytes filled with zero-bytes)
pointer map	0x01 or 0x02 or 0x03 or 0x04 or 0x05
locking page	0x00 (only, if db-size > 1 GB)

A page with links to other pages only is called a *b-tree interior* page [80]. The record nodes are saved in *table b-tree leaf pages*. Beyond this, a table can have multiple indexes. An index contains links to normal table records to speed up searching and sorting by specific fields. Whenever we create an index, SQLite creates a B-tree structure to hold the index data as well. Similar to normal tables we can distinguish between *index b-tree interior pages* as well as *index b-tree leaf pages*. When a data record is too large for a single data leaf page, the excessive bytes are spilt onto so-called *overflow pages*. Several overflow pages are filled at once to store large amounts of data such as Binary Large OBjects (BLOBs). Together all overflow pages for one record form a linked list. To capture all the data associated with a record, we need to read all the pages. The payload for an record and the preceding pointer are combined to form a cell.

Despite the five data page types, SQLite knows three more page classes. A database file might contain one or more pages that are not in active use. Whenever the last record is deleted from a page, this page is released. The freed page will be reused when new pages are required and filled with new table contents. In the meantime, all unallocated pages are stored in a so-called freelist (sect. 5.3). These freelist pages are of particular forensic value since most of the removed content can be found here.

A further not yet discussed page type are so-called *pointer maps*. A pointer map has the function of not losing track when pages are moved from one position in the database file to another. This page type is created whenever the database is reorganized or cleaned up. A pointer map provides a lookup table to quickly determine page types and their parents. However, this page type exists only in auto-vacuum databases. The *locking page* is the last page type in SQLite. The first page of this page class starts at byte offset 2^{30} (1,073,741,824) and always remains unused. Conversely, this means that a locking page only appears when the database size is more extensive the 1 GB. Since it is empty, it has only a technical, but no forensic value and is therefore not considered further.

We can usually determine the type of page by looking at the page's first byte. The flag-byte at offset 0 indicates the page class. Table 5.4 lists all the page types discussed so far. However, not every database will include all of these types. With the page size and type information at hand, an investigator can walk through the database and identify all areas of interest.

5.2.1 The Database Header

Every forensic investigation starts with analysing the file header. The header contains important information that will help us to carve for deleted records. The fields of the header have a precisely defined size and position (see Fig. 5.2). The individual (multi-byte) fields are encoded as big endian (BE) values.

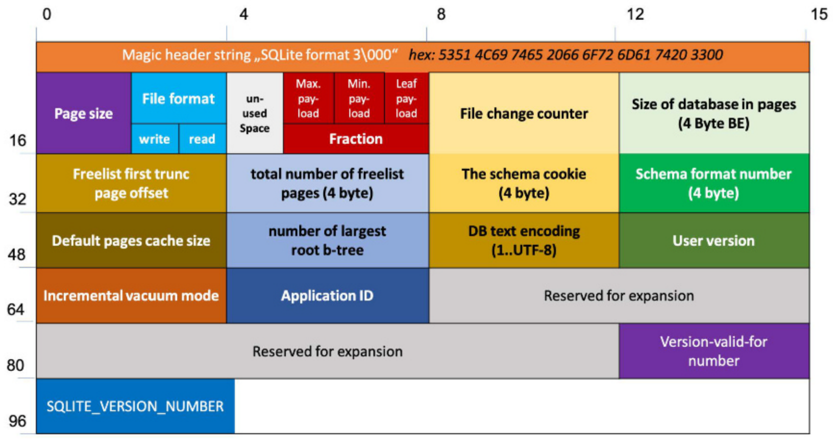

0
4
8
12
15
Magic header string „SQLite format 3\000" hex: 5351 4C69 7465 2066 6F72 6D61 7420 3300
Page size
File format
write
read
un-used Space
Max. pay-load
Min. pay-load
Leaf pay-load
Fraction
File change counter
Size of database in pages (4 Byte BE)
16
Freelist first trunc page offset
total number of freelist pages (4 byte)
The schema cookie (4 byte)
Schema format number (4 byte)
32
Default pages cache size
number of largest root b-tree
DB text encoding (1..UTF-8)
User version
48
Incremental vacuum mode
Application ID
Reserved for expansion
64
Reserved for expansion
Version-valid-for number
80
SQLITE_VERSION_NUMBER
96

Fig. 5.2: The SQLite Database Header Format and fields

We will discuss the fields below and evaluate them in terms of their respective value for a forensic investigation [80]:

- Each database starts with the header string. The magic header value is always set to "SQLite format 3". We can use the header information to carve the beginning of a database file on the binary level. Offset 15 marks the end of the magic header string. It holds a special character, the null terminator (0x00).
- At offset 16, we can find a two-byte big-endian integer value representing the database's page size. The value in this field must be a power of two. The range of values is between 512 and 32768. There is one exception: The value 0x0001 is viewed as a big-endian 1. It represents the value 65,536 - the largest possible page size - since this number will not fit in a two-byte usually.
- The two flag bytes at offset 18 and 19 control the read and write permission for the database. The values should typically always be either a 1 or a 2. For the rollback journalling mode (sect. 5.4.2), both values are set to 1. In contrast, number 2 in both fields indicates a WAL journalling mode (sect. 5.4.3). If the write version has a value greater than 2, this database file must be accessed as read-only. These two fields' value can indicate whether other files (WAL file or journal file) are present.
- The 1-byte integer value at offset 20 of the header is used to apply for certain SQLite extensions. The number of bytes specified here reduces the usable area within the page. In this way, for example, special salt or nonce values can be stored for each page when using the cryptographic extension. This value is usually 0. The value can be odd.
- The bytes on offset 21 to 23 have fixed values per definition. Maximum and minimum payload fraction must be 64, 32. The byte for the leaf payload fraction always holds the value 32.
- With each transaction carried out on the database, the 4-byte big-endian integer at offset 24 is usually incremented by one. A process that wants to read data from the database can determine whether there has been a change since the last access.
- With the 4-byte integer on offset 28 stores the size of the in-header database in pages. However, this value may differ from the file's actual size when accessing a database before version 3.7.0. Alternatively, you can determine the actual file size and divide by the page size to infer this value.
- At offset 32, we can find a 4-byte big-endian integer which indicates the beginning of the so-called freelist. As already pointed out, unused pages in the database file are stored within this data structure. This field has a significant meaning, as it allows us to access pages of the database that are no longer visible. It holds the offset of the first page of the list. If the value is zero, the list is empty.
- At offset 36 represents the total number of entries on the freelist. Together with the start address, one can thus automatically iterate over the released pages.
- Each change to the database schema, such as adding or deleting a table or creating an index, automatically leads to an increment of the value at offset 40.

- The 4-byte value at offset 44 represents the format number. This field has a value between 1 and 4. For a SQLite database created with the latest version of the database, the value is always 4 and thus supports the more SQL commands. Databases created before November 2005 usually have a value of 3 or less.
- The value default pages cache size at offset 48 queries or sets the suggested maximum number of pages of disk cache for a database file.
- The 4-byte big-endian integer value at offset 52 is only used to manage pointer-maps for auto vacuum-databases. A non-zero value means that this database file contains pointer map-pages.
- All strings in the database are encoded with the same encoding. There are only 3 valid encodings: UFT8 (value 1), UTF16LE (value 2), UTF16BE (value 3). For the analysis of the database, this field value must always be read first.
- The integer at offset 64 is true for incremental_vacuum and false for auto_vacuum mode. A value is larger than 0 means that the database reclaims space after data has been deleted. An autovacuum database thus contains few deleted artefacts - if any. It is defragmented automatically.
- The Application ID at offset 68 can be set by the Application programmer. It is not used by SQLite.
- Offset 92 covers the value of the change counter. The integer at offset 92 indicates which transaction the version number is valid for.
- The 4-byte integer at offset 96 stores the SQLITE_VERSION_NUMBER value. The version number of the database library with which changes were last made to the database is noted here.

All remaining header bytes are reserved for future expansion. Consequently, we can ignore them.

> Important

As can be seen from what has been said, various header fields must be read and analyzed as the first step of every examination. Thus, the page size (offset 16) and the number of pages (offset 28) must always be determined, since we need to know the structure and size of the database. In order to interpret the strings correctly, the encoding must also be examined (offset 58). A look at the freelist entries at offset 32 and 36 tells us whether unused pages in the database exist. If we do not find any references to free pages, it may be an auto vacuum database (offset 64). Using the flags for transaction management at offset 18 or 19, we can also find out which additional SQLite files may exist. This is of particular interest because these files can also contain records of former transactions. Thus, old states of the database have been overwritten in the meantime could be made visible again. The header's remaining information is more technical and is, therefore, less interesting for the investigator.

5.2.2 Storage Classes, Serial Types and Varint-Encoding

In order to understand the binary format of records we first need to clarify what data types SQLite knows at the binary level and how they are encoded. Like most other databases, SQLite uses strict typing. Therefore, each value stored is mapped to one of the five storage classes (Table 5.2). The word storage class is just another term for a data type. However, the latter is more commonly used in connection with programming languages. SQLite supports storage classes for integers (INTEGER), floating-point numbers (REAL), strings (TEXT), binary objects (BLOB), and other numeric data such as dates (NUMERIC). The storage class thus determines how the binary data is to be interpreted. Conceptually, each column of a table is assigned with a specific affinity. The affinity denotes the preferred storage class for a column. The data type of a column defines what value the column can hold. However, the SQL standard knows several data type names for one SQLite storage class. For example, there exist more than ten different integer data types in SQL. For texts, there exist seven different types. Accordingly, each data type is mapped to exactly one storage class.

A second essential aspect is a length occupied by a cell value. An integer, for example, will consume a length between zero and a maximum of 8 Bytes. A floating-point number is mapped to a 64-bit field. A text can have an arbitrary length. SQLite uses the so-called serial types to map storage class and length. In simplified terms, this type is a number. The concrete value of the number provides information about the length of a cell value. At the same time, the storage class can be derived from the numerical value. Table 5.3 lists all possible serial types. For serial types 0, 8, 9, the value is zero bytes in length. The serial type is used whenever the type and length of a cell must be determined. Usually, each table row has a corresponding header that summarizes the serial bytes for each column. As a rule, a serial type occupies exactly one byte. Especially with texts or BLOBs, this principle is sometimes deviated from as soon as the numerical value's length exceeds 127. In this case, additional bytes may be added to map the serial type.

Table 5.2: Mapping from SQL types to SQLite storage classes [80]

SQL Data Type	Storage Class
INT, INTEGER, INTUNSIGNED, LONG, TINYINT, SMALLINT, MEDIUMINT, BIGINT, INT2, INT8	INTEGER
TEXT, CHARACTER, CLOB, VARCHAR, NCHAR, NATIVE CHARACTER, VARYINGCHARACTER	TEXT
REAL, DOUBLE, DOUBLEPRESICION, FLOAT	REAL
NUMERIC, DEZIMAL, BOOLEAN, DATE, DTIME	NUMERIC
BLOB (no datatype specified)	BLOB

SQLite uses a particular encoding for storing serial types. The representation form used is a variable-length integer (varint). SQLite version 3 uses this simple byte-oriented encoding where each byte contains 7 bits of the integer being encoded. The most significant bit (MSB) is a flag bit, indicating more bytes to follow. Since most integers in a database have relatively small values, we can keep memory consumption low this way. Storing with a fixed-length integer will mostly generate unnecessarily many null bytes. Instead, SQLite uses a static Huffman encoding of 64-bit twos-complement integers that needs less space for small positive values. The serial type varints for large strings and BLOBs might extend up to nine-byte varints. The following illustration should once again make clear the storage principle of varint-values:

```
1 Byte     0XXXXXXXX                                   ..127
2 Bytes    1XXXXXXX 0XXXXXXXX                        ..16384
3 Bytes    1XXXXXXX 1XXXXXXX 0XXXXXXXX             ..2097152
4 Bytes    1XXXXXXX 1XXXXXXX 1XXXXXXX 0XXXXXXXX ..268435456
```

Since texts have a variable size, and the length calculation is performed by a formula. A numerical value above 12 or 13 can only occur with texts or BLOBs. An odd value will be correspondingly for texts. On the other hand, if the value is even, then it is the BLOB storage class. For example, to store the word *Test*, the value 21(0x15) - 2 * text length + 13 - is stored as the length specification. A JPEG file with, let us say, the length of 109 Bytes would be encoded with the serial type number 230 since $N * 2 + 12$ is what we need to calculate for a binary object. However, since we cannot map this value with 7 bits, we have to add a second byte for the varint:

```
decimal: 230 = 128 + 64 + 32 + 4 + 2
binary:  1110 0110
varint: 1000 0001 and 0110 0110 (2-Byte: 1X.. 0X..)
```

Thus, we must first calculate the respective length specification each time we need to know the exact length of a table cell. The serial values 8 and 9 are noteworthy features. They can be used to map the two values 0 or 1. An extra content byte is not necessary in this case. With the information presented, we are now able to decode the cells of a table row.

5.2.3 Decoding The SQLite_Master Table

A database schema is a set of data definitions that define the structural design of a database. As already explained, the schema, or the master table, resides on the database's first page, just behind the header. Technically, it is a regular table [85]. Table 5.4 shows all columns and their meaning for the master table. The schema table contains all database objects in the database and the statement used to create each object. With the schema table's help, all table names, the corresponding column names and data types can be determined. Each table entry is opened by two additional fields: the *rowid* and the payload (see Fig. 5.3). Both values are only visible on the

Table 5.3: Serial Type Codes Of The Record Format [80]

Serial Type	Size	Meaning
0	0	Value is a NULL.
1	1	A 8-bit twos-complement integer.
2	2	A big-endian 16-bit twos-compl. integer.
3	3	A big-endian 24-bit twos-compl. integer.
4	4	A big-endian 32-bit twos-compl. integer.
5	6	A big-endian 48-bit twos-compl. integer.
6	8	A big-endian 64-bit twos-compl. integer.
7	8	A big-endian 64-bit floating point number.
8	0	integer 0 (schema format $\geq$ 4).
9	0	integer 1 (schema format $\geq$ 4).
10,11	variable	Reserved for internal use. Variable size.
N$\geq$12, even	(N-12)/2	Value is a BLOB with (N-12)/2 bytes length.
N$\geq$13, odd	(N-13)/2	Value is a string in the text encoding and (N-13)/2 bytes in length. The nul terminator is not stored.

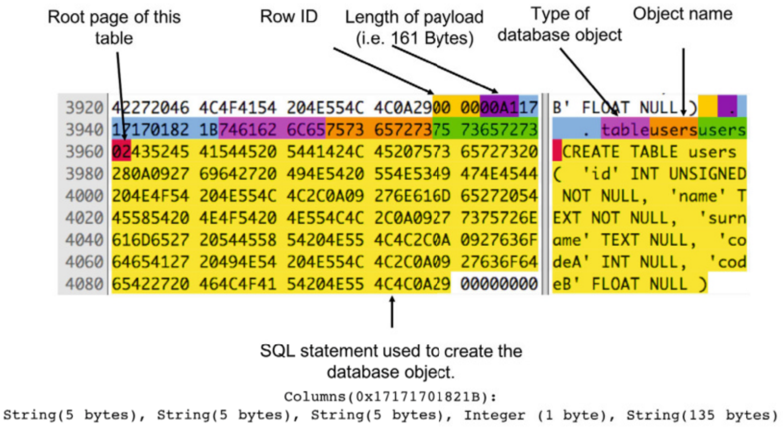
binary level. Any row of the master table and therefore every database object is assigned to a unique, non-NULL, signed 64-bit integer - the *rowid*. This value is used as the access key for the data in the underlying B-tree. On the binary level, each table row starts with a rowid number greater than null. Most tables in a typical SQLite database schema are rowid tables. A *rowid table* is defined as any table in an SQLite schema that is not a virtual table and is not a WITHOUT ROWID table. The rowid is not part of the table definition. A payload field that stores the length of the record follows directly after the rowid.

Table 5.4: Structure of the sqlite_master table [85]

Column Name	Description
type	type of database object (table, index etc.)
name	name of the database object
tblname	table that the database object is connected to
rootpage	root page
sql	SQL statement used to create the database object.

Interestingly, we can find descriptions for tables that have already been removed. If an object in the database is erased, the schema table's corresponding record is marked as removed. If a table is dropped, the rowid value for the line in question is set to 0x0000. The entry that is no longer needed is only overwritten when a new database object is added. In the meantime, the entry is still accessible. Figure 5.3 shows an example of a deleted entry for a table in hex mode. The table header and all columns of the record are intact. Only the rowid value at Offset 3935 has been wiped with zero bytes.

In the example below, the signature 0x7461626C65 represents the object type of a table. The table name, i.e. ”users”, directly follows the type column. However, we must parse and analyse the corresponding SQL statement from the fifth column to get all column names and the corresponding type information.


Root page of this table
Row ID
Length of payload (i.e. 161 Bytes)
Type of database object
Object name
SQL statement used to create the database object.

```
3920 42272046 4C4F4154 204E554C 4C0A2900 0000A117 B' FLOAT NULL ) .
3940 17170182 1B746162 6C657573 65727375 73657273    . tableusersusers
3960 02435245 41544520 5441424C 45207573 65727320  CREATE TABLE users
3980 280A0927 69642720 494E5420 554E5349 474E4544 (  'id' INT UNSIGNED
4000 204E4F54 204E554C 4C2C0A09 276E616D 65272054  NOT NULL,  'name' T
4020 45585420 4E4F5420 4E554C4C 2C0A0927 7375726E EXT NOT NULL,  'surn
4040 616D6527 20544558 54204E55 4C4C2C0A 0927636F ame' TEXT NULL,  'co
4060 64654127 20494E54 204E554C 4C2C0A09 27636F64 deA' INT NULL,  'cod
4080 65422720 464C4F41 54204E55 4C4C0A29 00000000 eB' FLOAT NULL )
```

```
Columns(0x17171701821B):
String(5 bytes), String(5 bytes), String(5 bytes), Integer (1 byte), String(135 bytes)
```

Fig. 5.3: Record of a dropped table from the sqlite_master (example)

By analyzing the SQL statement, a storage class can be derived for each table column. For the five columns of the table <users>, the following columns can be identified: INT, TEXT, TEXT, INT, REAL. This type of vector can be considered as a kind of fingerprint. Sometimes, a found record could be recovered, but it is not clear to which table it belongs. With the help of the table’s signature derived in this way, an assignment can still be made, even for a deleted record. Of course, this rule is not always 100% accurate. It is not excluded that two tables have the same signature. However, it can help us make an educated guess, which will be correct in most cases.

5.2.4 Page Structure

All records are stored on pages. Approaching the data of a table requires a leaf page scan. To access the data, we must understand the structure of a page. Each page starts with a header, with a total size of 8 bytes in the case of a data leaf page (see Fig. 5.4). All header bytes are big-endian values. The header starts with the page type at offset 0. In the case of a leaf page, the page starts with the value 0x0D. It can be classified from the other pages by reading this value. The 2-byte value at offset 1 marks the beginning of the first free block on the page. A free block is created whenever a record is deleted from the database. All free blocks are organized as a linked list, whereas the first two bytes of the free block point to the offset of the following free block within the list. If the free block is the last on the chain, this value is zero. If we want to identify deleted records, our search should start right here in the free block list [80].

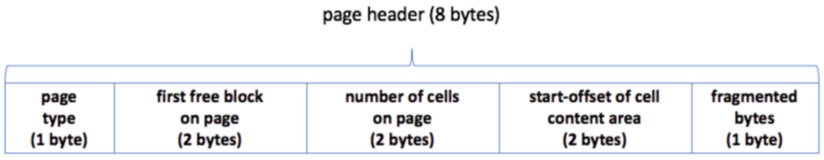

page header (8 bytes)
page type (1 byte)
first free block on page (2 bytes)
number of cells on page (2 bytes)
start-offset of cell content area (2 bytes)
fragmented bytes (1 byte)

Fig. 5.4: Fields of a b-tree leaf page header

Another essential value is located directly behind the free block field at offset 3. The 16-bit twos-complement integer field is called *number of cells*. Its value indicates how many active cells exist within the current page. In SQLite, the serial type header and the values of a particular table row are combined into a structure called "cell". So if we want to access a record, we need to locate the matching cell. Fortunately, all cell offsets are stored in an array directly after the page header. Hence, to read a regular record of a table, we need to iterate through the cell pointer field.

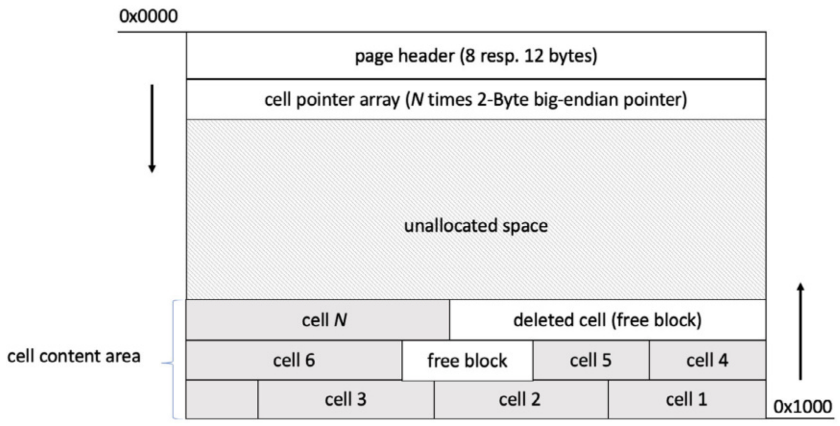
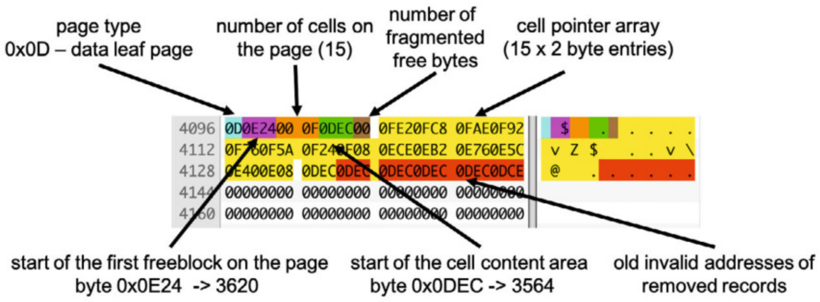

0x0000
page header (8 resp. 12 bytes)
cell pointer array (N times 2-Byte big-endian pointer)
unallocated space
cell N
deleted cell (free block)
cell content area
cell 6
free block
cell 5
cell 4
cell 3
cell 2
cell 1
0x1000

Fig. 5.5: Structure of a regular data leaf page (permanent and temporary)

The next header field at offset 5 provides the start-offset of the content area. A b-tree leaf page is divided into regions (see Fig. 5.5). The cell content area is always located at the bottom of the page. The header and the cell pointer array are always located at the beginning of the page. Between them resides the unallocated space. As the content area grows from the highest memory address towards the lower address, overlapping the two mentioned regions is prevented. The concept is thus similar to the management of heap and stack areas within memory management. The last value in the header denotes the number of fragmented bytes. A free block requires at least 4 bytes of space. Areas between 1 to 3 bytes form a fragment and thus cannot hold any data records.

Figure 5.6 shows an example of the header of a page on a binary level. In addition to 15 cells, we can also find at least one free block of offset 3620(0x0E24). The content area in this example starts at 0x0DEC. The cell pointer array is highlighted in yellow. Interestingly, we can find five more cell pointers shown in red. The value of the surplus cell offsets corresponds to the start offset of the cell content area. From this, we can conclude that apparently, five other records must have existed on the page in the past. Nevertheless, they have been deleted in the meantime. Thus, in addition to the 15 regular records, there should be five more deleted records on the page. However, the deletion turned the cells into free blocks. So, to find and restore them, we need to examine each element of the free block list.



```
4096  0D0E2400 0F0DEC00 0FE20FC8 0FAE0F92  $ .  . . . .
4112  0F760F5A 0F240F08 0ECE0EB2 0E760E5C  v Z $  . . v \
4128  0E400E08 0DEC0DEC 0DEC0DEC 0DEC0DCE  @   . . . . . .
4144  00000000 00000000 00000000 00000000
4160  00000000 00000000 00000000 00000000
```

page type
0x0D – data leaf page
number of cells on the page (15)
number of fragmented free bytes
cell pointer array (15 x 2 byte entries)
start of the first freeblock on the page byte 0x0E24 -> 3620
start of the cell content area byte 0x0DEC -> 3564
old invalid addresses of removed records

Fig. 5.6: Sample header and cell content array for a data leaf page

It is not always possible to find all deleted records by checking the free blocks. If a record is deleted that resides directly at the unallocated area, the offset value for the cell content area start is moved up in the direction to a higher address. Of course, this address denotes the cell pointer offset of the next regular record. The data set is thus moved to the unallocated area by changing the border. We must consider this case in our search since this record will never appear in the free block list.

However, it gets even worse. If a complete page is deleted, SQLite typically wipes the first 4 Bytes of the header with zeros. So, in this case, the offset for the first free block is erased. Thus, we do not know where precisely the list begins. What does this, in turn, mean for our search for hidden records? The best way to approach our search for slack areas is to use the exclusion principle. *Slack space* is the leftover storage that exists on a page when records do not need all the space which has been allocated. Slack areas are always created when records are deleted. Hence, the total amount of slack space can thus be calculated as shown in the equation below.

```
slack space with (possible) deleted content = page content
                       - header (8 bytes)
                       - N times 2-Byte cell pointer
                       - fragmented bytes
                       - N times cell
```

If we exclude the regular, well-known areas of the page, we automatically access the slack areas. Only the areas determined in this way can contain deleted data artefacts. In any case, we must always consider the unallocated space and the free block list when searching within the page. Fortunately, leaf pages are always structured the same. However, there is a second type of leaf page, the index leaf page. In its structure, this page corresponds to a regular data leaf page, except for one difference. The index leaf page starts with the value 0x0A at offset 0. However, what has been said so far also remains valid for the second type of page.

5.2.5 Recovering Data Records

Now that we know the location of the records, we can start reading them. This information can be derived from the cell offset array (see the last section). Every cell has the same structure (see Fig. 5.7). The cell header opens with a payload value. It indicates the total size of the cell in bytes. This value does not include the cell header itself. Normally, the payload field is followed by the rowid (see sect. 5.2.3). As already explained, the pseudo-column is usually generated automatically by SQLite. It is used to enable efficient access via the table tree. However, not all records have a rowid. For example, index records are created without this field. If the option "WITHOUT ROWID" is part of the CREATE TABLE statement, this field is also missing. Thus, the cell header has a minimum size of 1 byte for a mandatory payload value. The values in the cell header and all other header fields are varint values without a fixed size. So to read a record, we always have to read value by value. Skipping or omitting bytes is not possible because the fields do not have a fixed offset. The actual cell starts again with a header. This time, it is the header of the data record.

The *header size* field indicates how many bytes the header contains. Its value includes the actual header size byte. The individual serial types follow immediately. Column by column, we must first determine the storage class and space for each table cell. The header is followed directly by the actual data record. Since we operate on a binary level, the exact length of each field to be read and the data types can only be determined via the serial bytes in the header. However, it might be challenging to determine the exact beginning or end of the column cell values without this information. An intact header is, therefore, an essential prerequisite for successful data recovery.

> **Information**

The recovery of deleted data depends on the data management policy used. This, of course, differs from application to application. We can distinguish three cases:

1. **Wipe with zeros.** The free block is completely overwritten with zero-bytes. Recovery of data is impossible even if the removed area is identified.
2. **Truncate or remove deleted area.** The second policy is made on a small size of data. It deletes the record itself, and there is no way even to trace the occurrence of deletion. Some iPhone system files are handled this way.
3. **Add to a free list.** The last policy is to mark the record or page as free. The data itself remains in the database. This procedure generates the least I/O-traffic compared to the other two strategies. It is therefore used as the default behaviour of SQLite.

In the case of a data record that has been deleted, it sometimes happens that the cell header and parts of the record header are replaced with new information [59]. These new data fields cover the free block's length in bytes and the address of the following free block. Since both pieces of information are mapped to a 16-bit fixed-length integer, a total of four bytes of the respective cell are overwritten. In total, we can discern six situations when dealing with a deleted record (see Table 5.5).

Many records are deleted without being marked or overwritten. As explained earlier, some records are deleted by merely moving the cell content area's border upwards. Thus, the records slip into the unallocated area of the page. When clearing the browser cache, for example, almost all entries are removed from a caching table. Instead of first marking each record as deleted, the links to the affected pages are deleted from the table tree. Anything else would be a time-consuming process. Instead, the page as a whole is skipped. In both cases, however, the deleted records remain intact. Complete reconstruction is, therefore, possible. Sometimes a record is removed from the middle of the content area of an active page. In this case, the record

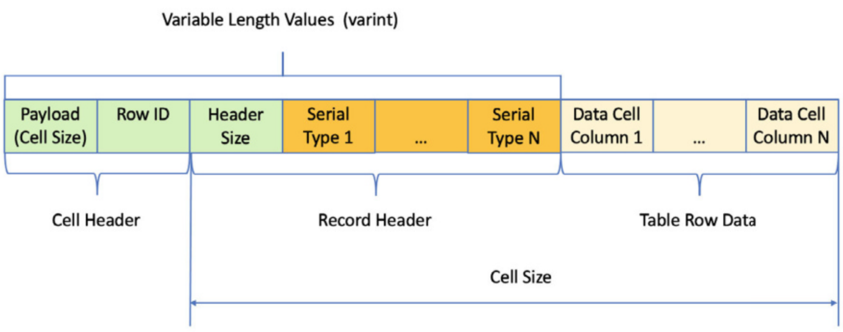


Fig. 5.7: Schematic structure of a data leaf cell

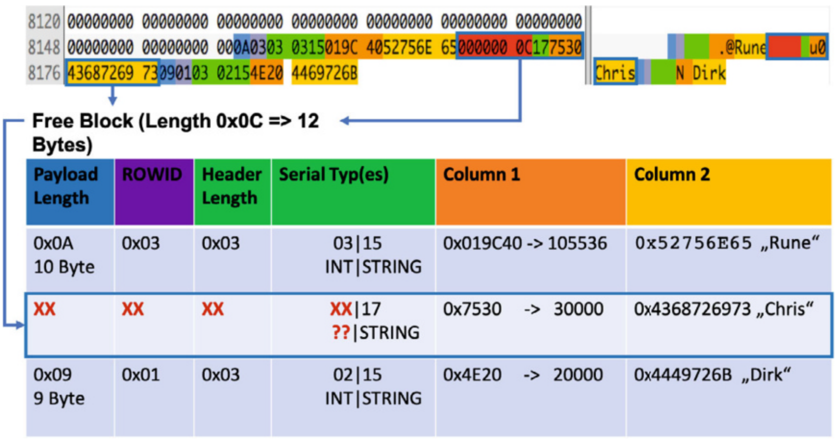
Table 5.5: Recovery Situations

Wiped Data	Recoverability
cell is intact (no wiped bytes)	yes
payload bytes	yes
payload bytes + rowid	yes
payload bytes + rowid + header length	yes
payload bytes + rowid + header length + 1st serial	partly
two or more serial type are wiped	no

is converted to a free block. Thus, the beginning is overwritten, at least partially. The previously occupied space will be released for reallocation. This, in turn, can result in different cases that influence the recoverability of the data record. Sometimes only the payload got wiped. In another case, the payload field, together with rowid, may be overwritten. We can mostly do without this field information. As long as the rest of the cell record remains intact, we can read the required column lengths and types and correctly interpret the data. Even a wiped header length field should not be a big problem. This field only holds the total length of the header. It can be reconstructed by summing the individual serial lengths. It gets tricky when columns are also overwritten. Without a valid column type and length specification for our first column, we cannot reconstruct the remaining columns correctly. However, the first column of a table is often an ID column with a numerical value. Knowing the length of the first column of a regular record on the same page can indirectly infer the first column's length for our destroyed record. Unfortunately, this rule does not work in every case. For example, if the first column contains a text with variable length, we will most likely not restore the record correctly. If more than one serial type has been overwritten, reconstruction seems unlikely. We then have too many possible lengths to consider. Strictly speaking, the number of possible lengths for a column grows exponentially with the number of overwritten length or type information in the header.

Figure 5.8 shows the content area of a data leaf page. There are a total of three records on the page. The cells are located at the end of the page. Remember, the cell content area always grows from higher towards the lower address. The record in the middle is deleted. The records before and after it are intact. Cell header, record header and all data are unaltered. Even without knowledge of the table, it can be deduced from the serial types alone that it is a table with apparently two columns. The first column can store integers (serial types $0x02$ resp. $0x03$). The second column is a string since the value is odd and greater than 13 (see sect. 5.2.2).

We can see that the second of the three data cells have been deleted because the first 4 bytes of the data set have been overwritten with the free block identifier. The identifier is $0x0000000C$. The first two bytes have the value $0x0000$. From this, we can conclude that it is the last free block within the page. The second half of the identifier tells us something about the length of the free block. It is exactly 12 bytes ($0x000C$). The free block is outlined in red in the illustration. As we can see, the actual



```
8120 00000000 00000000 00000000 00000000 00000000 00000000 00000000
8148 00000000 00000000 000A0303 0315019C 4052756E 65000000 0C177530   .@Rune u0
8176 43687269 73090103 02154E20 4469726B                            Chris N Dirk
```

Payload Length	ROWID	Header Length	Serial Typ(es)	Column 1	Column 2
0x0A 10 Byte	0x03	0x03	03|15 INT|STRING	0x019C40 -> 105536	0x52756E65 „Rune“
XX	XX	XX	XX|17 ??|STRING	0x7530 -> 30000	0x4368726973 „Chris“
0x09 9 Byte	0x01	0x03	02|15 INT|STRING	0x4E20 -> 20000	0x4449726B „Dirk“

Fig. 5.8: Example data page with three records (one is wiped)

data fields of the deleted record are still intact. However, the PPL-field, ROWID, header length byte, and the first column's serial type are no longer accessible. The serial type of the second column is not wiped. From the length specification of the free block and the knowledge about the length of the second column, we can infer the length of the first column in this case. Accordingly, the first column of our data set can only be 2 bytes in size:

```
length of the first column field =
         12 byte (total free block length)
        - 5 (0x15 -13 / 2) (length of text column)
        - 4 (free block identifier)
        - 1 (serial type byte for 2nd column)
```

Thus, we can recover deleted content in many cases, even when parts of the header have been overwritten.

5.3 Accessing The Freelist

As soon as the last record on a page is deleted, it is transferred to the free list. At the same time, the link within the table tree is removed. From now, the page cannot be accessed from an active table. However, it can be assigned to a new table at any time. Meanwhile, the content of the page is still accessible. Usually, it is not wiped or replaced with random values. The pages are just sitting on the free list, waiting to be used again. Like the slack areas in the standard database pages, these unused pages may contain forensically exciting values such as chat protocols, short messages, or web pages visited [61].

The freelist is a simple linked list consisting of *trunk pages* 5.9. Each trunk page initially contains a 4-byte integer pointer referencing to the next trunk page in the list

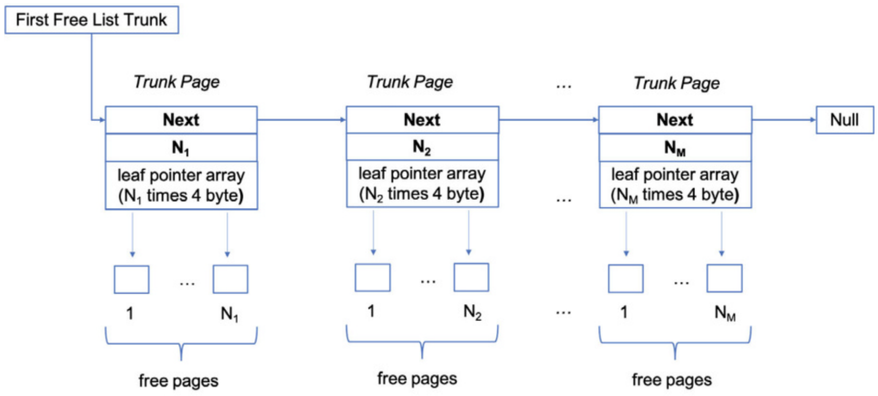


Fig. 5.9: Schematic principle of a freepage trunk list

[80]. A zero-byte value means that this is the last trunk page in the list, and the list ends here. The second 4-byte value in a trunk page contains the number of leaf page offsets. To analyse all the list pages, we must first visit each trunk page and query the offsets stored. Nevertheless, where do we have to start our search for freelist treasures?

The starting address for the freelist can be calculated very easily [59]. We must first determine the start offset of the first trunk page from the header at offset 32 of the database. Second, we need the page size. The latter can also be determined from the header. From these two values, we can calculate the actual offset of the first trunk page:

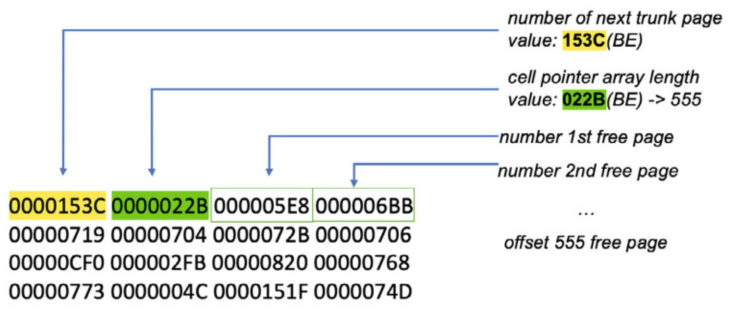
```
offset of 1st trunk page = (trunk page number - 1) * page size.
```

A trunk page consists of an array of 4-byte big-endian integers. As pointed out, the first 4 bytes of the trunk page header references the next trunk page within the list. The next, a four-byte big-endian integer holds the length of the leaf pointer array of the current page. With these two pieces of information at hand, we can quickly iterate over the array's entries.

The basic algorithm is shown in Listing 1. An example of a trunk page will illustrate what has been said so far (see Fig. 5.10). In addition to the reference to the next trunk page at offset 0, the number of page pointers to follow is visible (offset 4). The offset of the first free page can be found directly behind the two header integers at Offset 8. The second pointer is exactly 4 bytes behind. In the example, there are a total of 555 entries on the TrunkList page. The data size is therefore 8 + 555 * 4 = 2228 bytes. Thus, all unused pages can be found and accessed with linear time complexity with the described algorithm.

Algorithm 1 Freelist Page Recovery

```
                                                        ▷ Input: SQLITE $db$ filepointer
 1: read $pagesize$ ← 4 byte BE on byte 0x10
 2: read trunk ← for the first freelist trunk on byte 0x20
 3: while trunk ≠ null do
 4:     $start$ = (4 Byte BE in offset - 1) * $pagesize$.
 5:     $db$.seek($start$)                                    ▷ go to start of the trunk page
 6:     read trunk ← for the next freelist trunk page (4 Byte BE)
 7:     read $length$ ← number of cell entries (4 Byte BE)
 8:     for $j = 0, 1, \ldots, length - 1$ do                 ▷ iterate over trunk page array
 9:         $db$.seek($start + 8 + (4 * j)$)
10:         read freepage ← next free page number
11:         fpstart = (freepage - 1) * $pagesize$.
12:         $db$.seek(fpstart)                                ▷ go to start of next free page
13:         readPage()                                        ▷ start analyzing the hidden page
14:     end for
15: end while
```



```
0000153C 0000022B 000005E8 000006BB
00000719 00000704 0000072B 00000706
00000CF0 000002FB 00000820 00000768
00000773 0000004C 0000151F 0000074D
```

Fig. 5.10: start of a freelist trunk page (example)

5.4 More Artefacts

As explained earlier, SQLite manages all records in a single database file. However, access management, transaction handling, and integrity protection are performed with the help of primarily additional temporary files [84]. Despite the main database file, SQLite uses nine distinct types of temporary files (see Fig. 5.11). Below we will take a look at the other file types of SQLite. The focus is on searching for records no longer in the regular database but can still be found in one of those files.

5.4.1 Temporary File Types

SQLite creates several temporary files when managing the database. A *transient database*, for example, is a temporarily created file when the database is reorganized.

Data pages that are no longer required are removed. The whole process is comparable to the defragmentation of a hard disk. Pages are joined together, and gaps are closed. Then, the temporary file's content is copied back into the original database file, and the temporary file is deleted. However, this file type is generated only for databases for which the VACUUM property is activated. Since the database copy is deleted immediately afterwards, it is not easy to locate it on the disk. However, it might be possible to find old page versions of the database on the medium through carving. From time to time, SQLite makes use of *transient indices*. Each index is therefore stored in a separate temporary file. For example, if the ORDER-BY or GROUP-BY clause is used in an SQL statement, a corresponding index file is created to manage the intermediate results. The index is automatically deleted at the end of the statement that uses it.

In the case of complex SQL statements, partial queries are sometimes stored in a temporary file. In SQLite, this method is called "materializing" the subquery. This is the case, for example, with large SQL INNER JOIN statements. The query optimizer decides for which query a separate swap file is created.

Database users can create a temporary table using the "CREATE TEMP TABLE" command. Since this unique table is created only for a particular database connection and is not visible to other database users, it is swapped out to a separate file. Again, the temporary database file used to store temporary tables is removed automatically when the database connection is closed. When SQLite performs a transaction with multiple statements, a *Statement Journal File* can be used to undo individual steps. Assume that by executing a statement, 100 rows of a table are modified. After half of the records have been modified, the execution must be aborted due to an error. The rows of the database that have been modified so far are written back with the statement journal's help. All five of the temporary file formats discussed can contain data or temporary results of the database transactions. However, these data are highly volatile. In most cases, the temporarily stored results are already deleted when the statement is finished. Thus, it is not very likely for an investigator to come into contact with such artefacts. We will, therefore, not consider them further.

There are four remaining file types in SQLite. Unlike the formats discussed so far, these are files that are often encountered when examining a database. These files are *Rollback Journals*, *Write-ahead Logs*, *Shared-Memory Files* as well as *Super Journals*. They can usually be found in the same directory as the actual database file. Admittedly, the data stored in it is also classified only temporary within the official documentation of SQLite. However, the data stored in them is updated or overwritten much less frequently. We almost always find one of these file types. For this reason, these are also listed under the heading *other permanent files* in Fig. 5.11. Thus, the chance to acquire data from these files is much more likely. However, in some cases, the use of one file format excludes the use of the second. For example, the shared memory file and write-ahead log are usually found together. In contrast, the rollback journal is only found in a directory if the first-mentioned files are absent. Of the file formats mentioned above, super journals are relatively rare. The files are created only in transactions where multiple databases are updated simultaneously in an atomic transaction. Accordingly, without a super-journal in place, transaction commit on

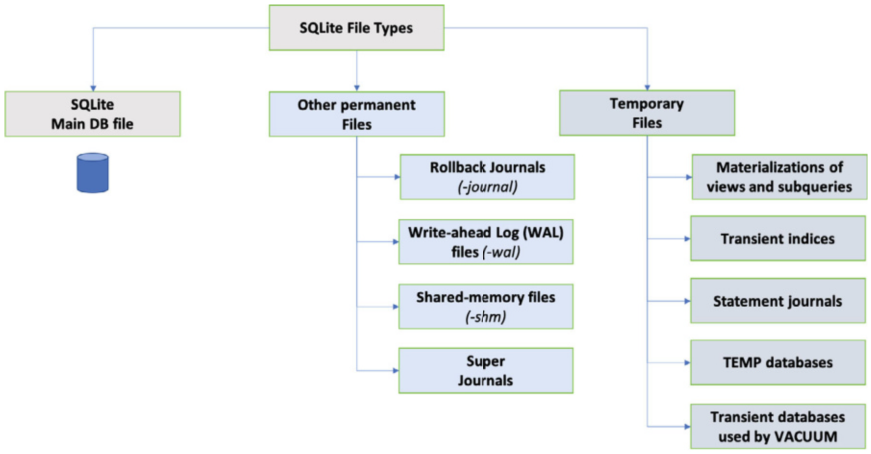

SQLite File Types
SQLite Main DB file
Other permanent Files
Temporary Files
Rollback Journals (-journal)
Write-ahead Log (WAL) files (-wal)
Shared-memory files (-shm)
Super Journals
Materializations of views and subqueries
Transient indices
Statement journals
TEMP databases
Transient databases used by VACUUM

Fig. 5.11: The SQLite file types (permanent and temporary)

a multi-database transaction would be atomic for each database individually, but it would not be atomic across all databases. Due to the relatively low usage level, we will not take a closer look at this file format. Instead, we will focus on the two remaining journal formats. Besides availability and confidentiality, data integrity forms a central goal of every database system. SQLite is no exception. SQLite maintains its integrity by using journals and transactions. Below we will examine the two integrity protection techniques offered by SQLite in more detail: Write-ahead logs and Rollback Journals [80],[32].

5.4.2 Rollback Journals

The idea behind the rollbacks is simple: If a database gets into an inconsistent state due to write access, it is reset to the last valid state. To implement atomic commit and rollback capabilities, SQLite offers a file called *rollback journal*. Rollback refers to resetting the individual processing steps of a database transaction [79]. The system is thus wholly returned to the state before the start of the transaction. In the case of SQLite, a copy is first created for all database pages possibly affected by the transaction and stored in the rollback journal. If something goes wrong during transaction processing, the database can always be reset to the last valid state if required. Note: SQLite permanently stores the entire page in the journal file, even if the transaction modifies only a single record.

A journal file is usually created when a new transaction is started and deleted after the transaction is completed. Although this is the default behaviour, in many cases, there is a deviation from this approach. For example, if the application developer activates the *exclusive locking mode* for a database, then the rollback journal is not immediately deleted. An application can enable the exclusive locking mode by using the following pragma-statement:

```
PRAGMA locking_mode=EXCLUSIVE;
```

In this case, the journal file may be truncated, or the file's header may be wiped with zero bytes. Which behaviour of this occurs depends on the SQLite version used. However, the file is preserved in any case as long as the locking mode is activated. Fortunately, many applications that use rollback journals for transaction safety operate in this mode, reducing unnecessary IO operations. The same behaviour as is seen in EXCLUSIVE locking mode can also be reached by setting the *journal mode pragma* to PERSIST instead of DELETE which is the default behaviour in SQLite:

```
PRAGMA journal_mode=PERSIST;
```

No matter which of the two modes is activated, an investigator can restore the old execution states of the database. In this way, data records that may have been deleted in the meantime can be made visible again.

! Attention

The rollback journal file is always located in the same directory as the actual database. One can quickly identify the journal by the file name: It has the same name as the database but with the extension **"-journal"**. Thus, the name of a journal file is precisely eight characters longer than the original name of the database [84].

A rollback journal is a binary format. Just like the main database file, it contains a small header. The header has a fixed size of a maximum of 28 bytes. The individual header fields and their meanings are shown in Table 5.6. Next to the Magic Header String, information about the total number of database pages stored in the journal. The header also records the original size of the database file. So if a change causes the database file to grow, we will still know the original size of the database. Unfortunately, the fields carried in the header are usually automatically overwritten after a COMMIT and wiped with null bytes. Thus, we will rarely be able to recover useful information from it. However, the header is usually preserved if a transaction cannot be completed due to a power down.

The journal file has a preset page size. The value can be determined via the offset 20 in the header. Even if this value can no longer be determined due to wiping, there is a way out. The default value of the first sector is 512. The remaining space of the first journal page is filled with zero bytes. Since the default page size is 512 bytes, the header is thus always followed by a padding area of zero bytes. After the header and padding area, zero or more page records will follow. Such a record contains a copy of precisely one database page. Additionally, each record is introduced by a one-field header. Only with this value, SQLite can reset the correct page in the database in case of a rollback. On offset four, the original content of the database

Table 5.6: Rollback Journal Header Format

Offset	Size	Description
0	8	Header string: 0xd9, 0xd5, 0x05, 0xf9, 0x20, 0xa1, 0x63, 0xd7
8	4	The "Page Count" - The number of pages in the next segment of the journal
12	4	A random nonce for the checksum
16	4	Initial size of the database in pages
20	4	Size of a disk sector.
24	4	Size of pages in this journal.

page follows. The journal page record ends again with a 4-byte big-endian value. It holds the checksum for this page. The value is used to guard against incomplete write operations.

Table 5.7: Rollback Journal Page Record Format [80]

Offset	Size	Description
0	4	The page number in the database file
4	N	Original content of the page prior to the start of the transaction
N+4	4	Checksum

Since the header is always reset for each new transaction, the page records directly following the header are always the most current. However, journal records of past transactions can still be stored in the same journal. For example, suppose a transaction changed ten database pages. The following transaction only rewrote five pages. In that case, the database subsequently contains the database's state before the last transaction plus five more pages from the previous. The following example shows the beginning of the second journal page of a rollback file:

```
|0x1200|61746506 BAC4E54E 0000000B 0D000000 |ate....N........|
|0x1210|0B0E2C00 0F620F35 0FC20F96 0F0E0EFD |..,..b.5........|
|0x1220|0ED10EB8 0E9A0E5A 0E2C0000 00000000 |.......Z.,......|

0xBAC4E54E  -> Checksum of the 1st journal records
0x0000000B  -> page 11 in the database (start of the 2nd journal)
0x0D000000  -> start of a data leaf page (snapshot)
```

The start of 2nd journal record can be calculated as follows:

```
  0x0200   1st sector (header + padding area) - 512 byte
+ 0x0004   page record page number (record start) - 4 byte
+ 0x1000   1st page in journal - 4096 byte
+ 0x0004   checksum of 1st journal page (record end) - 4 byte
 -------
  0x1208   start offset of the 2nd journal record
```

The example shows the end of the first journal page and the beginning of the second journal frame. While the green highlighted value at offset 0x1204 still belongs to the first journal page, the value at offset 0x1208 already initiates the next journal record. Generically, the address of each journal could be determined as follows:

$$\texttt{Record}_{start}\texttt{(N+1) = size of 1st sector + N} \times \texttt{(page size + 8)}$$

However, how can we determine whether the database's journal page belongs to the last transaction or is not perhaps older? A different random nonce is used each time a transaction is started to minimize the risk that unwritten sectors might by chance contain data from the same page that was a part of prior journals. The last nonce is a 4 Byte integer value and can be found at offset 12 in the journal header. By changing the nonce for each transaction, stale data will still generate an incorrect checksum. Since the entire page is always saved from the database, we can restore the actual data described in section 5.2.5.

5.4.3 Write-Ahead Logs

As pointed out in the last section, a copy of the data page to be changed is first created before writing directly into the database file in a classic rollback journal [86]. Version 3.7.0 of the SQLite database engine introduced an alternative concept for transaction management [84]. With *write-ahead logs (WAL)*, this procedure is reversed. The content of the original database file is not changed. Instead, every change is appended into a separate WAL file. It works like a roll-forward journal. All changes are first written to the WAL file. Even a COMMIT does not automatically update the database file [79]. If, for example, other reading database connections exist simultaneously, they can operate as usual on the original unaltered data. Meanwhile, a concurrently running write process stores its changes into the WAL file. Moving the WAL file transactions back into the database is called a *checkpoint*. Usually, SQLite does a checkpoint automatically. If the WAL size reaches a threshold size of 1000 pages, a checkpoint is triggered by default. As soon as we examine a database that works in WAL mode, we must also analyse the included WAL archive. Simultaneously, this also means that we may have different versions of the same database page in the main database and the WAL file. As long as no checkpoint has been carried out, the WAL file exclusively contains the latest changes. The database is, therefore, still in an old state. If we look at both files together, we can get a consistent view [86].

To access the content of a WAL file, all we have to do is open the corresponding database file. When opening a WAL mode database, the WAL file's content is automatically transferred back to the database. In other words, a checkpoint is executed. However, this procedure is usually not recommended for various reasons. With this approach, old artefacts that are evidentially valuable to the investigator could be overwritten and thus lost. Moreover, we would be violating a fundamental rule of any forensic investigation: Never change the evidence.

Important

It is best not to work with a standard database viewer when evaluating a database in WAL mode. Even by opening the database, one risks losing old data due to checkpointing.

But how should we proceed then? One possibility is the use of a special forensic database browser. An example would be the FQLite[1] browser. This program reads the database and the WAL file separately. Since access is read-only, all data is preserved.

Attention

A particular database will use either a rollback journal or a write-ahead log. It is not possible to use both at the same time. The write-ahead log is always located in the same directory as the actual database. One can quickly identify the journal by the file name: It has the same name as the database but with the extension **"-wal"**.

Let us now turn to the actual structure of the file. The WAL file starts with a header. Zero or more so-called WAL-frames follow it. Just as with the rollback journal, a frame represents the altered content of exactly one page of the database. The file header has a size of exactly 32 bytes. It starts with a 4 byte long Magic Number (see Table 5.8). At offset 4 follows the file format version. Again, this is a 4-byte unsigned integer value. The size of one page of the database is stored at offset 8. Using the field *checkpoint sequence number* at offset 12, we can again determine how many checkpoints have already been executed since their creation.

Table 5.8: WAL Header Format [86]

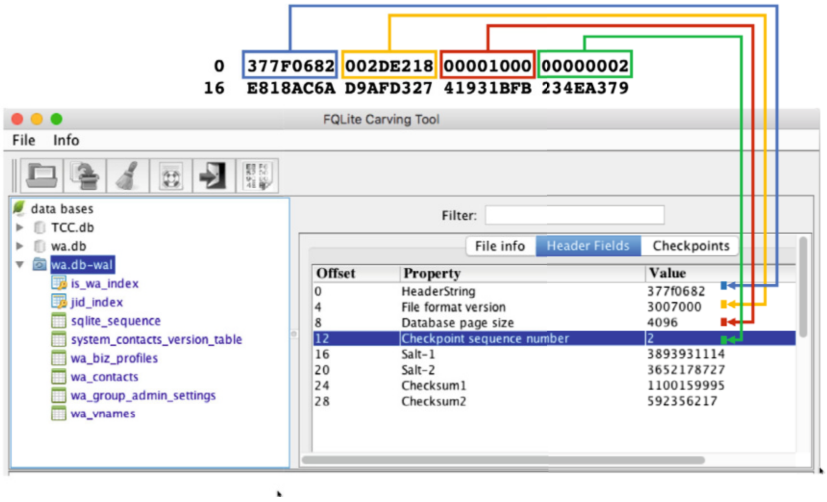
Offset	Size	Description
0	4	Magic number. 0x377f0682 or 0x377f0683
4	4	File format version. For example 3007000.
8	4	Database page size. Example: 1024
12	4	Checkpoint sequence number
16	4	Salt-1: random integer incremented with each checkpoint
20	4	Salt-2: a different random number for each checkpoint
24	4	Checksum-1: First part of a checksum on the first 24 bytes of header
28	4	Checksum-2: Second part of the checksum on the first 24 bytes of header

The last fields of the header form two salt values and two checksum values. Using these fields, we can determine which frames belong to the current checkpoint and

[1] https://github.com/pawlaszczyk/fqlite

have not yet been transferred to the database. Figure 5.12 shows an example of the header of a WAL archive in FQLite.

Each WAL frame also starts with a header [84]. The structure of the header with its fields is shown in Table 5.8. The header consists of exactly six big-endian values, each with a size of 4 bytes. The first object is the page number this frame is assigned. Using the page number, we can identify the place in the database where the change takes effect. The value at offset four can be used to determine whether a COMMIT was performed. A value other than 0 is a so-called *commit frame*. Let us remember that a COMMIT does not automatically update the database. Like the header of the WAL file, each frame header ends with two salt values and two checksums. The four big-endian 32-bit unsigned integer values are located from Offset 8 to 24.



```
 0  377F0682 002DE218 00001000 00000002
16  E818AC6A D9AFD327 41931BFB 234EA379
```

Offset	Property	Value
0	HeaderString	377f0682
4	File format version	3007000
8	Database page size	4096
12	Checkpoint sequence number	2
16	Salt-1	3893931114
20	Salt-2	3652178727
24	Checksum1	1100159995
28	Checksum2	592356217

Fig. 5.12: View on a WhatsApp-DB WAL-Header with FQLite Carving Tool

A WAL archive always grows from the beginning. It can cause frames from different checkpoints to appear in the same file, whereas current ones are always at the file's beginning. Fortunately, we can use the mentioned salt values to determine relatively quickly whether the frame under consideration is valid or whether it belongs to an older state already transferred to the database. Whether a frame is valid can be determined as follows [86]:

1. The Salt-1 and Salt-2 values from the header must both match the values in the respective frame.
2. The 8-byte checksum in the frame must match the cumulative checksum over the first 24 bytes of the WAL header plus the first 8 bytes and the contents of all previous frames.

If a checkpoint was executed successfully, the WAL file is reset afterwards. In this case, the salt values are overwritten. The value of salt-1 is incremented, while a

Table 5.9: WAL Frame Header Format [86]

Offset	Size	Description
0	4	Page number
4	4	For commit records, the size of the database file in pages after the commit. For all other records, zero.
8	4	Salt-1 copied from the WAL header
12	4	Salt-2 copied from the WAL header
16	4	Checksum-1: Cumulative checksum up through and including this page
20	4	Checksum-2: Second half of the cumulative checksum.

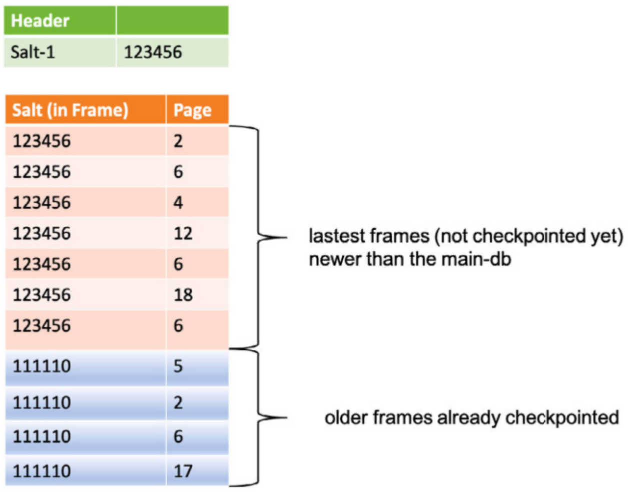
new random value is assigned to salt-2. Previously valid frames are automatically discarded due to this procedure. However, the previous frames usually remain in the archive due to the I/O- operations when the file is truncated. Thus, there is an excellent chance to make past states of database pages visible again with the help of the WAL file.

Let us take a look at how write-ahead logs work. Figure 5.13 shows the frames list of a WAL file. Below the header field for the Salt-1 value, several frames are shown. All frames with matching salt values belong to the same checkpoint. The first seven frames thus form a unit. The remaining frames are part of an older checkpoint. As we can see, the salt in the header matches the salt in the first unit. Accordingly, the pages have not yet been transferred to the database. In other words, the WAL file contains the latest version of page 2,4,6,18. The pages within the database are out of date. The next checkpoint is usually executed when opening the database, and these data records are transferred to the database. Since WAL files always work at a page level, the complete database page is updated. Remember, the salt value changes for each checkpoint. Thus the Salt-1 field in the header is discarded afterwards.

Interestingly, page 6 has been updated three times. When a checkpoint occurs, each page will be written back to the database in the same order written to the WAL file. Pages are written from the start of the WAL file. Accordingly, the update order would be 2,6,4,12,6,18,6. This allows a timeline to be created, starting with the first to the last update step.

5.5 Conclusions

The SQLite database format has great importance in the field of mobile forensics. In this chapter, we have therefore tried to take a look behind the scenes. As quickly became apparent, the file format of SQLite has some similarities to a classic file system, where files are usually stored in blocks. Instead of blocks or clusters, data content in SQLite is managed in pages. As has been shown, even records are often recoverable after they have been deleted. Analogous to a file system, these are usually



Header	
Salt-1	123456

Salt (in Frame)	Page
123456	2
123456	6
123456	4
123456	12
123456	6
123456	18
123456	6
111110	5
111110	2
111110	6
111110	17

lastest frames (not checkpointed yet)
newer than the main-db
older frames already checkpointed

Fig. 5.13: Frame list of a WAL file (example)

not wiped but merely marked as deleted. However, we do not manage files but data sets.

We further identified different slack spaces of an SQLite database. Besides free blocks and the unallocated space, we can find deleted records, especially in the freelist area of the database. The carving techniques discussed within this chapter can help make these data sets visible again in many cases.

Of the temporary file-formats considered, rollback journals and the WAL files are of particular interest to the investigator, as they may contain old or previously altered data. However, special care must be taken when acquiring data from these files. Thus, the data stored in a WAL file can be reconstructed manually or with specialized forensic tools. Using an ordinary SQLite reader, on the other hand, can lead to the loss of data.