

Missing SQLite Records Analysis

By: Shafik G. Punja & Ian Whiffin

Background

The SQLite database engine is one of the most widely used database formats, where its use can be found in countless areas such as web browsers, instant messengers, all smartphones, Mac computers, Windows 10 computers, also automotive infotainment systems, and surprisingly also found in smart television sets and cable boxes [1]. The utilization of SQLite databases across a wide spectrum of so many mediums, is due to its performance, reliability, portability, simplicity and accessibility of data. SQLite can be used as on disk application file format [2], or as an SQLite Archive (where the SQLite Archive is similar to a ZIP file or archive or Tarball) [3].

This article will specifically discuss the identification of missing records, within the SQLite database in its use as an application file format. The various analysis tools that will be used to analyze missing records within SQLite databases will be noted throughout the article. The authors are working from the premise that recovery of deleted, partially recoverable, or wholly intact recoverable records, is no longer viable. What will not be covered is the explanation on the various methods to recover deleted records. For that we direct you to the only textbook on this subject [authored in 2018 by Paul Sanderson, titled, SQLite Forensics](#).

SQLite Database Structure Basics

The SQLite database as a file on disk can consist of 3 separate files. There are, however, actually nine distinct types of temporary files that can be used by SQLite during database processing operations [5]. But for the purposes of this article, we will briefly mention the database file itself and the 3 types of temporary files ('shm', 'wal', and 'journal') that are most commonly encountered by digital forensics practitioners.

1. **The database file itself.** It can use various types of suffixes (listed below), or in some cases the SQLite database file will not have any suffix appended after the arbitrary prefix file name.
 - o '*.db', '*.db3', '*.sql', '*.sqlite', '*.sqlite2', '*.sqlite3', '*.sqldb'
 - o Example SQLite database filename: 'sms.db'.
2. **The database write-ahead log (WAL) or journal file.** The use of either a WAL or roll back journal file is determined by the value within the SQLite database file at decimal offsets 18 and 19. The legacy refers to the use of a journal file. Both types of files (WAL and journal) serve the same purposes of [atomic commit](#) and rollback, but both implemented in different ways. In addition to several other system level benefits over journal files, WAL files tend to perform faster [6].

Database Header Format		
Offset	Size	Description
0	16	The header string: "SQLite format 3\000"
16	2	The database page size in bytes. Must be a power of two between 512 and 32768 inclusive, or the value 1 representing a page size of 65536.
18	1	File format write version. 1 for legacy; 2 for <u>WAL</u> .
19	1	File format read version. 1 for legacy; 2 for <u>WAL</u> .

Figure 1: Screenshot sourced from: <https://www.sqlite.org/fileformat2.html#vnums>

- If a WAL file is used by SQLite, then a digital forensics practitioner can potentially observe three files as shown in the screenshot below.

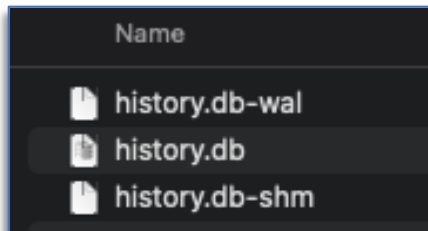


Figure 2: history.db with WAL and SHM files

- If the legacy rollback journal file is used by SQLite, then a digital forensics practitioner, can potentially observe two files as shown in the screenshot below. Note the absence of the shared memory (*.db-shm) file, as this type of temporary file is only used with WAL.



Figure 3: main.db with journal file

- **WAL file:** The WAL file can be found in filesystems where the database connection has not shutdown cleanly. This allows for **potential recovery** of live records that have not been committed to the database, or recovery of previously existing records that have been deleted and are no longer recoverable from the database itself, but all or part of the record resides in the WAL file. The WAL file can be recognized by the presence of 4 characters, "-wal" appended to the end of the name of the main database filename [5]. Example WAL filename: 'sms.db-wal'.
- **Rollback Journal file:** This rollback journal file provides another method of 'atomic commit' and rollback method for SQLite database operations. An SQLite database journal file can be identified by the 8 characters "-journal" appended to the end of the filename [5]. Example 'journal' filename: 'sms.db-journal'

3. **The shared memory ('shm') file.** This file **does not contain any database content**. If a database crash occurs, the 'shm' file is not vital to the recovery of the database [4]. In the authors' experience, the shared memory file has not been observed to be of investigative value. Example 'shm' filename: 'sms.db-shm'.

Database Header Structure, which is comprised of the first 100 bytes, can be thoroughly enjoyed by referring to this source: <https://www.sqlite.org/fileformat2.html#vnums>

The SQLite database file header (or magic number) is 16 bytes in length, starting at decimal offset 0 (zero), can easily be recognized in ASCII as "SQLite format 3" (or in hexadecimal as 53 51 4C 69 74 65 20 66 6F 72 6D 61 74 20 33 00).

The Record Number Autoincrements

The SQLite database itself consists of tables, where each table has rows of records with columns that describe the data within the rows. The SANS SQLite Pocket Reference Guide (created Lee Crognale, Heather Mahalik and Sarah Edwards) has a nice graphic that visually explains this [10].

Tables	ZUSER	ZRECEIVEDTIMESTAMP	ZTIMESTAMP	ZBODY
ZKIKATTACHMENT (184)	37	497800782.987472	497800782.474	Welcome to Kik, the super fast smartphone messenger! If you have any questions, let me know. I'll do my best ☺
ZKIKATTACHMENTEXTRA (178)	41	497803811.6537	497803811.6537	You started chatting with Ace
ZKIKATTRIBUTION (3)	41	497804389.586069	497804196.154	Hey lloyd, so glad we're finally in touch
ZKIKCHAT (5)	41	497805068.863391	497805067.896	7cbf883b-8672-44e0-97fe-c3705e75f7c7
ZKIKCHATEXTRA (5)	41	497805068.863391	497805067.896	WHAT do you think of this picture?
ZKIKMESSAGE (558)	41	497805068.863391	497805067.896	WHAT do you think of this picture?
ZKIKMESSAGEEXTRA (4062)	41	497805068.863391	497805067.896	WHAT do you think of this picture?
ZKIKPUBLICGROUPS (1)	41	497805068.863391	497805067.896	WHAT do you think of this picture?
ZKIKUSER (292)	41	497805118.132724	497805118.132724	I just sent you one of my current laptop
ZKIKUSEREXTRA (292)	41	497805427.726313	497805427.726313	Hello microphone
Z_4MESSAGES (558)	41	497805442.399056	497805442.399056	Test chat from ace to lloyd
Z_9ADMININVERSE (5)	41	497812324.589263	497812324.156	I saved a kik picture too
Z_9BANSINVERSE (0)	41	497813709.744746	497813693.037	I saved the pic of the trash and then I deleted the pic of the trash
Z_9MEMBERS (32)	41	497813764.51298	497813764.51298	I saved the pic of the trash and then I deleted the pic of the trash
Z_METADATA (1)	41	497904591.777452	497904590.659	3cba5c30-d11d-456c-960a-a1ac7de8672f
Z_MODELCACHE (1)	37	498070317.634271	498070317.634271	Hi kik

Figure 4: SANS SQLite Pocket Reference Guide (created by Lee Crognale, Heather Mahalik, and Sarah Edwards)

As stated on the 'sqlite.org' website: 'All rows within SQLite tables have a 64-bit signed integer [primary] key that uniquely identifies the row within its table'. In other words, by default, every SQLite database has a 'rowid' column (or alternately may choose to use the PRIMARYKEY as a PK column) that uniquely identifies a record within a specific row in the database itself [7]. Record numbering starts with a value of 1, and autoincrements with every new record that is created. The record numbers are therefore contiguous, with no gaps [8].

****Note: The reader should also be aware, there also exists the ability within an SQLite database to create a row, using 'WITHOUT ROWID' table method, that omits the creation of a 'rowid' column [7]. The 'WITHOUT ROWID' table method is not default behaviour for an SQLite database.****

The following screenshot is an example of the 'sms.db' SQLite database file (from Cellebrite CTF of Ruth Langmore's Apple iPhone X) opened **with the 'sms.db-wal'** present, using [DB Browser for SQLite](#). This a great free open source cross platform application that can open an SQLite database file. The 'sms.db' SQLite database file Apple devices running iOS 14.x (or iPadOS 14.x) and lower, records SMS, MMS and iMessage activity for the native Messages application.

The messages table shows the ROWID column which contains a total of 73 records. In the screenshot record numbers 1 through to 20 are displayed in a contiguous manner. There are no missing records from rows 1 to 73. We checked.

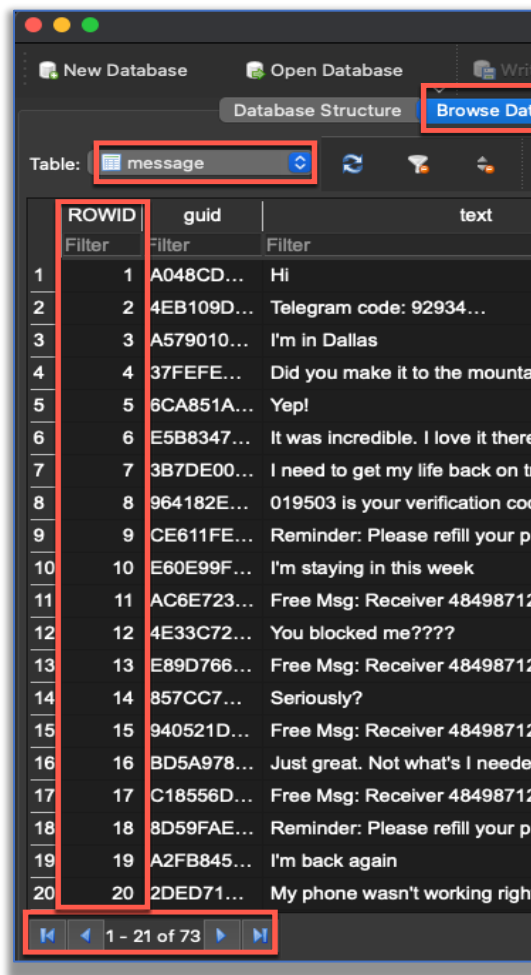


Figure 5: Screenshot from [DB Browser for SQLite v 3.12.1](#) on macOS 11.0.1, showing contiguous record numbers in the ROWID column.

If the 'sms.db' SQLite database file (from Cellebrite CTF of Ruth Langmore's Apple iPhone X) is opened with and without its associated WAL file, 'sms.db-wal', there is a difference in the number of contiguous records present in the messages table. Two copies of the 'sms.db' SQLite database were created in separate storage locations. The first copy did not have the associated WAL file, 'sms.db-wal' but the second copy did. Both 'sms.db' SQLite database files were opened in their own instances of [DB Browser for SQLite](#) as shown in the screenshots below.

The 'sms.db' SQLite database file **without** its associated WAL file, 'sms.db-wal', presents with **65 contiguous records in the messages table** with ROWID column values from 1 through to 65.

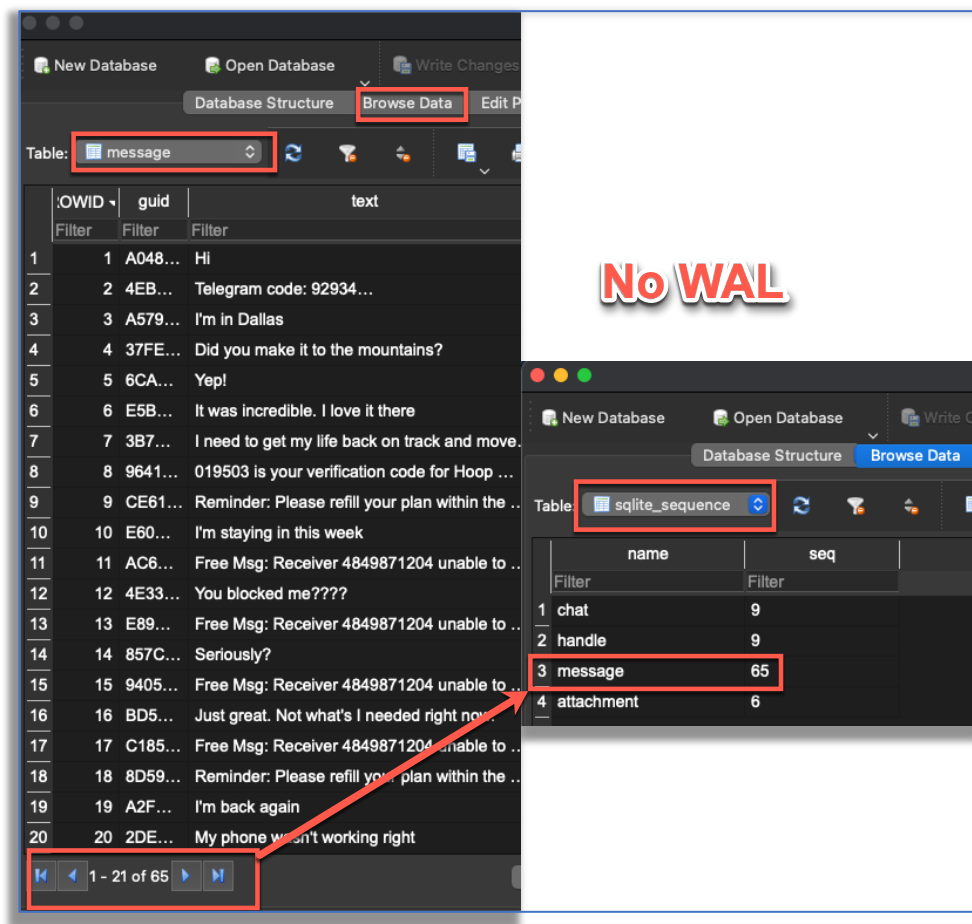


Figure 6: Screenshot from [DB Browser for SQLite](#) v 3.12.1 on macOS 11.0.1, no WAL.

The 'sms.db' SQLite database file **with** its associated WAL file, 'sms.db-wal', presents with **73 contiguous records in the messages table** with ROWID values from 1 through to 73. Compared to the previous, that is a difference of 8 live records that were not committed (or merged) to the 'sms.db' SQLite data, until it was opened with the associated WAL file, 'sms.db-wal' present.

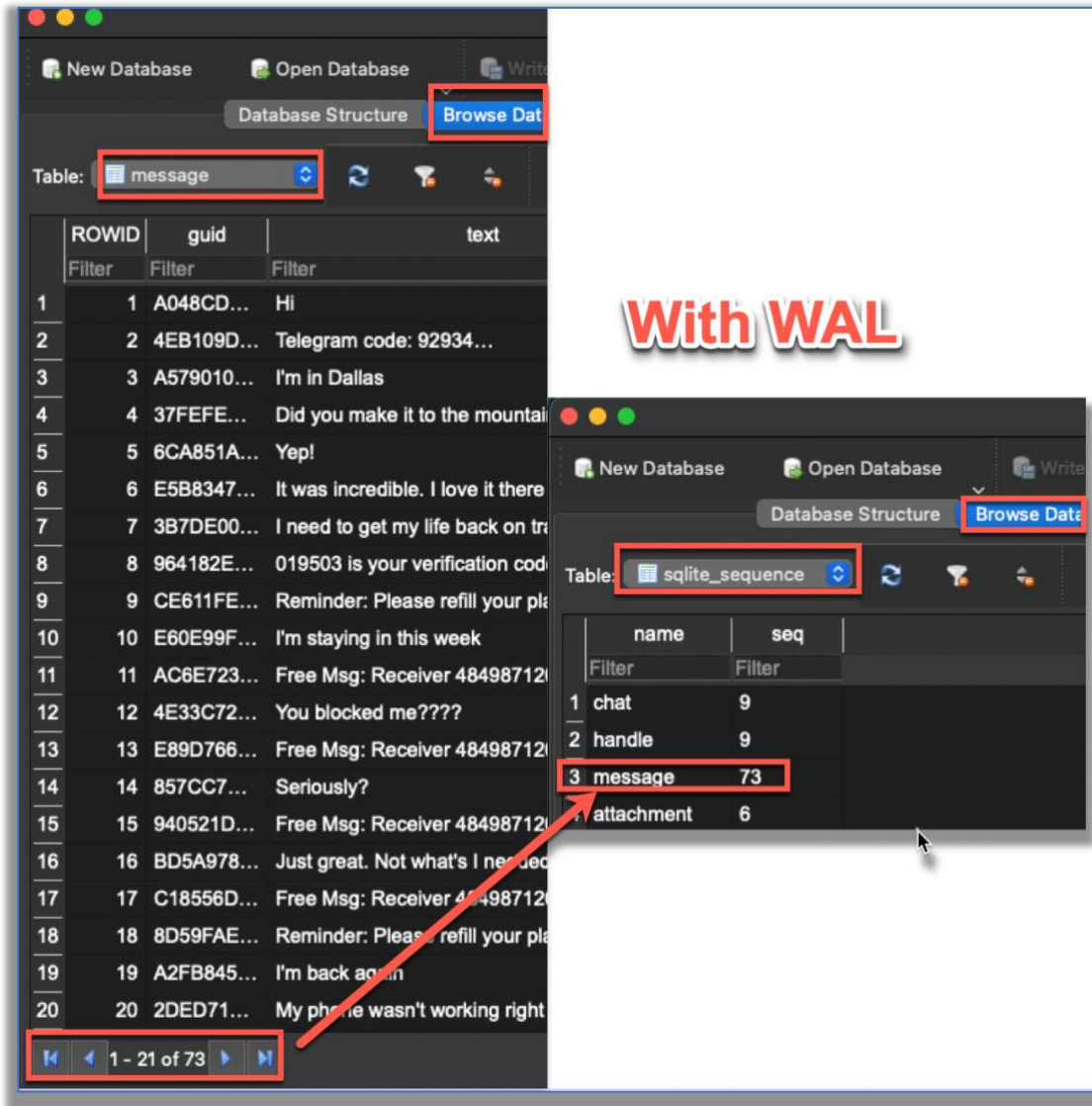
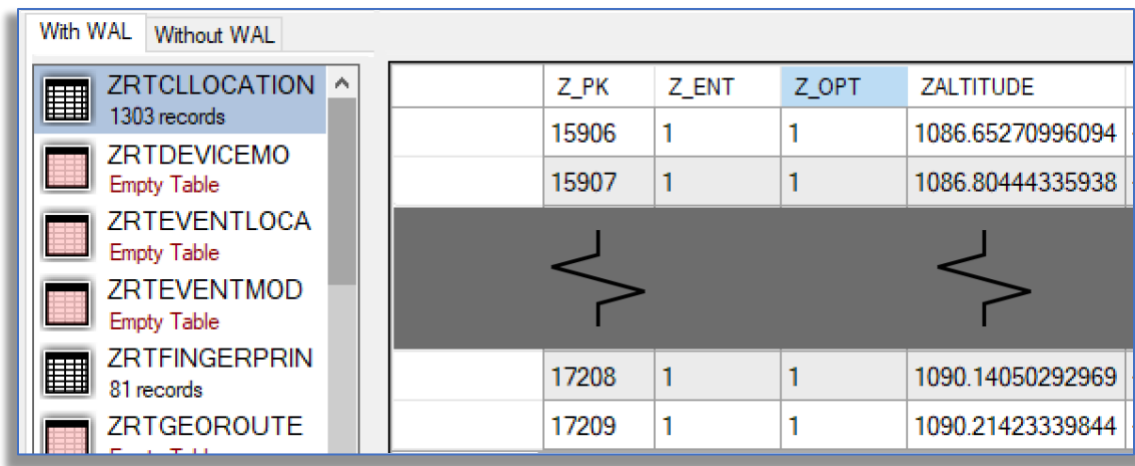


Figure 7: Screenshot from [DB Browser for SQLite v 3.12.1](#) on macOS 11.0.1, with WAL.

So, in this example, how important is the 'sms.db-shm' file? When the 'sms.db' SQLite database file is opened with its associated WAL and shm files present, the same number of records is observed as with WAL file only.

It is important to note also that the inverse can be true. The main SQLite database may contain data that gets removed once the WAL has been incorporated. The reason behind this is because there could be records that are assigned to be deleted from the main SQLite database, for which the write (delete record) operation has not been committed. The WAL contains all the pending write operations and as the delete operation is also a write, you will find them together with other writes in the WAL. So, in this case opening the main SQLite database with the WAL file will cause the records to be removed.

To demonstrate this, we opened up a cache database from an iPhone both with and without processing the WAL file using using [ArtEx](#) (developed by Ian Whiffin). When the WAL was processed, you can see that the ZRTCLOCATION table contains 1303 records and that the first record is PK 15906 and the last PK is 17209, in the screenshot below.

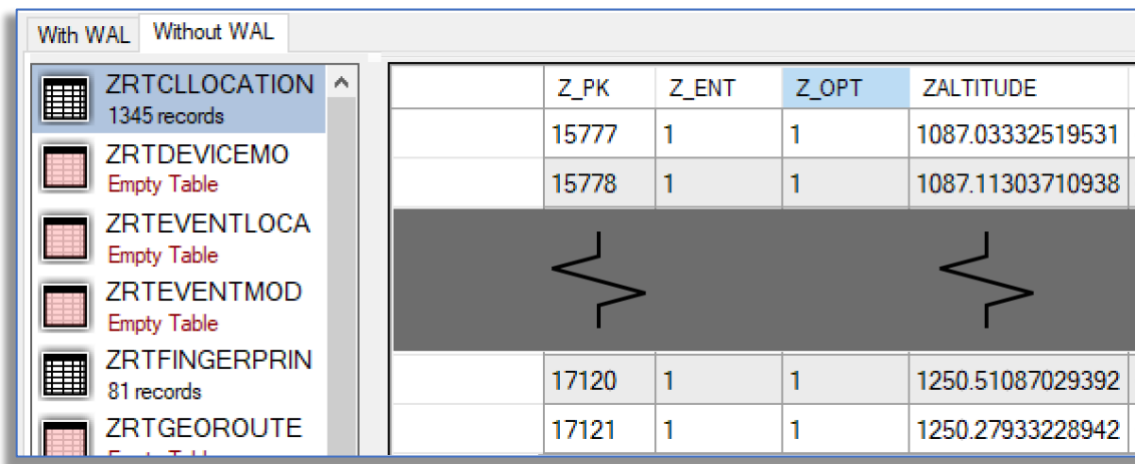


The screenshot shows the ArtEx interface with the 'With WAL' tab selected. The ZRTCLOCATION table is highlighted in the left sidebar, showing 1303 records. The main table view displays the following data:

Z_PK	Z_ENT	Z_OPT	ZALTITUDE
15906	1	1	1086.65270996094
15907	1	1	1086.80444335938
~			
17208	1	1	1090.14050292969
17209	1	1	1090.21423339844

Figure 8: Screenshot of ArtEx with WAL parsing.

Compared to without WAL parsing as shown below. When the WAL is not processed, you can see there are 1345 records, with a first and last PK or 15777 and 17121 respectively.



The screenshot shows the ArtEx interface with the 'Without WAL' tab selected. The ZRTCLOCATION table is highlighted in the left sidebar, showing 1345 records. The main table view displays the following data:

Z_PK	Z_ENT	Z_OPT	ZALTITUDE
15777	1	1	1087.03332519531
15778	1	1	1087.11303710938
~			
17120	1	1	1250.51087029392
17121	1	1	1250.27933228942

Figure 9: Screenshot of ArtEx without WAL parsing.

This makes perfect sense. When the WAL commit occurs, records 15777 through to 15905 are deleted (a loss of 129 records) but records 17122 to 17209 are created (a gain of 88 records). This leaves us with a net loss of 41 records. But the net difference really isn't the point (After all, a single record could make all the difference when it comes to evidence). The point is that many tools automatically process the WAL (if present) and will therefore risk the loss of data by record deletion. **But you cannot afford to ignore the WAL as it may contain new records that you may need.**

Once the connection to the 'sms.db' SQLite database is closed, both the 'sms.db-wal' file and 'sms.db-shm' files, were observed to be removed from the locations they were copied to, as the changes have been committed or merged within the 'sms.db' file.

The good news for digital forensics practitioners, is that most (if surely not all) digital forensics products, incorporate the associated WAL (or the rollback journal) file, as part of the analysis process, in order to identify as many unmerged live, unique records as possible. The authors' in this context, are not taking into account the recovery of deleted records from the freelist space specific to the SQLite database or from within the WAL (or the rollback journal) file itself. What we are unable to say is exactly which digital forensics products clearly identify, unmerged live records from the WAL

If the digital forensics practitioner elects to export a SQLite database file, out of their chosen digital forensics analysis platform, for review in any free open source tool, as demonstrated above, it is important to understand the associated WAL (or the rollback journal) and to know if you want to include it or not. As a caution to the reader, any free open source tools also have the ability to cause write changes to the SQLite database, as opposed to those SQLite database analysis tools that are part of a digital forensics tool suite.

Deleted SQLite Records

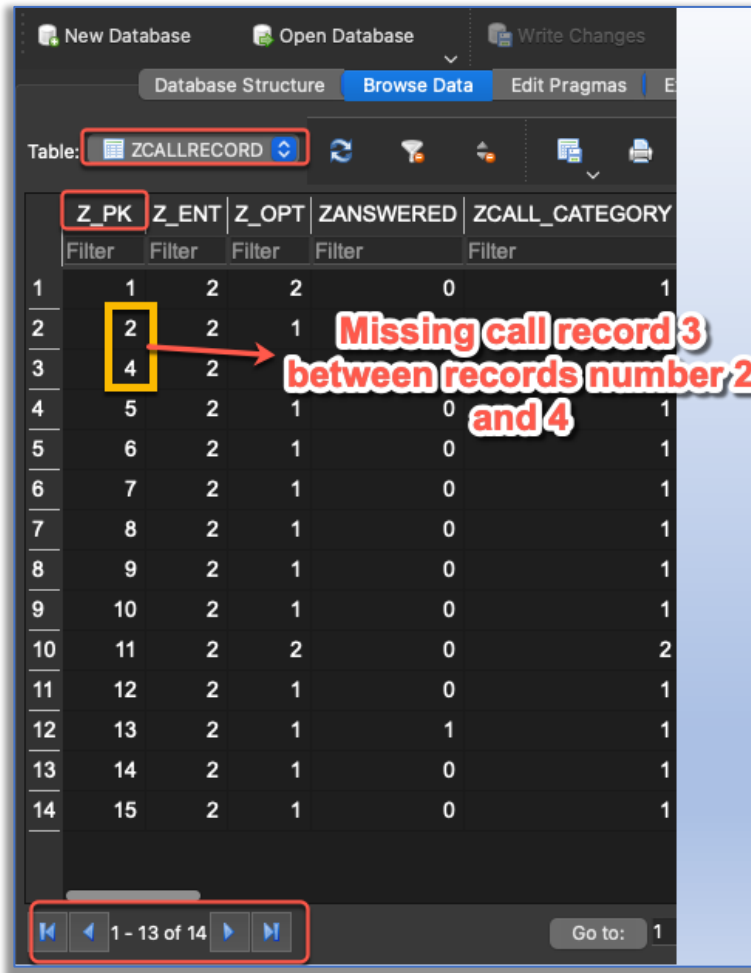
As smartphone operating systems evolve, the ability to recover deleted records from SQLite databases becomes increasingly more difficult. Generally speaking, until the SQLite database is fully vacuumed and defragmented, deleted (not live) records can be recovered. Once the vacuum and defragmenting operation occurs (transparent to the device user), deleted (not live) records are permanently removed, at which point there is no hope in any record recovery [11].

Take for example pre Apple iOS 12, where recovery of deleted records related to text messages and iMessages is more likely, as the deleted records were not immediately wiped. As of iOS 12 and newer the deleted text messages and iMessages are wiped almost immediately after deletion, and cannot be recovered from the SQLite database [11].

When a record is deleted, regardless of whether it can be recovered or not, the contiguous numbering sequence is broken, and gaps start to appear in the ROWID numbers. This leads to the actual point of the article, which is the analysis of the missing records from a specific table within an SQLite database file.

Missing SQLite Records

The identification of non-contiguous numbering sequences in the record numbers, within an SQLite database can be useful for identifying a missing record. For example, a missing record in the SMS database may signify a deleted message or a missing record from the call history table may signify a missing call record.



	Z_PK	Z_ENT	Z_OPT	ZANSWERED	ZCALL_CATEGORY
1	1	2	2	0	1
2	2	2	1		
3	4	2			
4	5	2	1	0	1
5	6	2	1	0	1
6	7	2	1	0	1
7	8	2	1	0	1
8	9	2	1	0	1
9	10	2	1	0	1
10	11	2	2	0	2
11	12	2	1	0	1
12	13	2	1	1	1
13	14	2	1	0	1
14	15	2	1	0	1

Figure 10: Screenshot from [DB Browser for SQLite](#) v 3.12.1 on macOS 11.0.1 showing a missing call record in the *CallHistory.storedata* database (from Cellebrite CTF of Ruth Langmore's Apple iPhone X).

For tables with timestamps, it may be useful to work out the time period that the deleted record was made. As the records are created consecutively, it stands the reason that the records are created in time order. Therefore, the missing record was made after the previous record but before the next. This may not always be the case and should be applied on a table-by-table basis.

Also note that this refers to the time that the record was made in the database, not necessarily the time the record was deleted or even the time that something occurred. SMS for example may be delayed in transit, and so the time received may be much later than the time it was sent.

This has been addressed in a blog post authored by Ian Whiffin titled, “[Primary Key / Date Stamp fallacy](http://doubleblak.com/blog/primarykeyfallacy)” (<http://doubleblak.com/blog/primarykeyfallacy>).

You may also want to take this further and explore other associated tables to see if there is any correlation you can make with the time period of the missing record(s). For example, are there any notifications, application use, sounds etc. made that could give you additional clues about the missing record.

You are unlikely to get as much information from a missing record as you would like; but that doesn't mean you can't get anything. This technique is easy when you have an obviously missing record, but sometimes the missing record isn't quite so obvious.

In the screenshot shown below, in the CallHistory.storedata database, ZCALLRECORD table , (from Cellebrite CTF of Ruth Langmore's Apple iPhone X) the missing record is actually **AFTER** the last record being displayed. It isn't immediately apparent as there is no record after it.

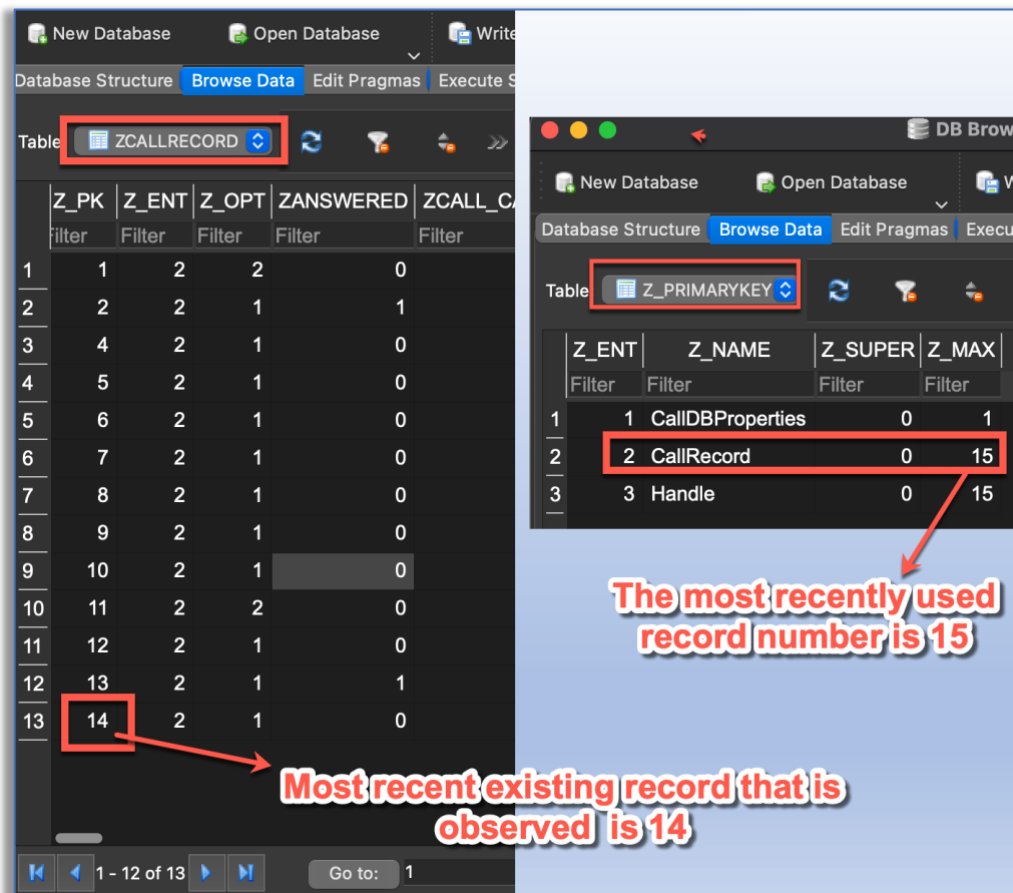


Figure 11: Screenshot from [DB Browser for SQLite](#) v 3.12.1 on macOS 11.0.1 showing a missing record after the last used record in the CallHistory.storedata database (from Cellebrite CTF of Ruth Langmore's Apple iPhone X).

What we have to do in this case is try to find the last generated record value in use within the table of interest. In this case, the Z_PRIMARYKEY table helps identify the last generated record value in use as record 15, and the last observed record in the Z_PK column, in the ZCALLRECORD table is 14. The reader should note that the authors intentionally deleted the last record, number 15 (row 14) to demonstrate the last record number used value can be one higher than the observed record number in its respective table.

In most cases, you may find a table called “sqlite_sequence” as an internal table within the database. The sqlite_sequence table is an internal table maintained by the SQLite database. It is automatically created by the SQLite database when there are fields in tables that have the AUTOINCREMENT keyword set [12].

Now let’s try and locate the “sqlite_sequence” table. One approach is to query a list of all the tables present within an SQLite database. The following custom query will list all tables present in the SQLite database:

```
SELECT name FROM sqlite_master WHERE TYPE = 'table'
```

Using the ‘sms.db’ SQLite database file (from Cellebrite CTF of Ruth Langmore’s Apple iPhone X) with its associated WAL file, a list of tables is returned, where the name of each table is being returned, and presented in non-alphabetical order.

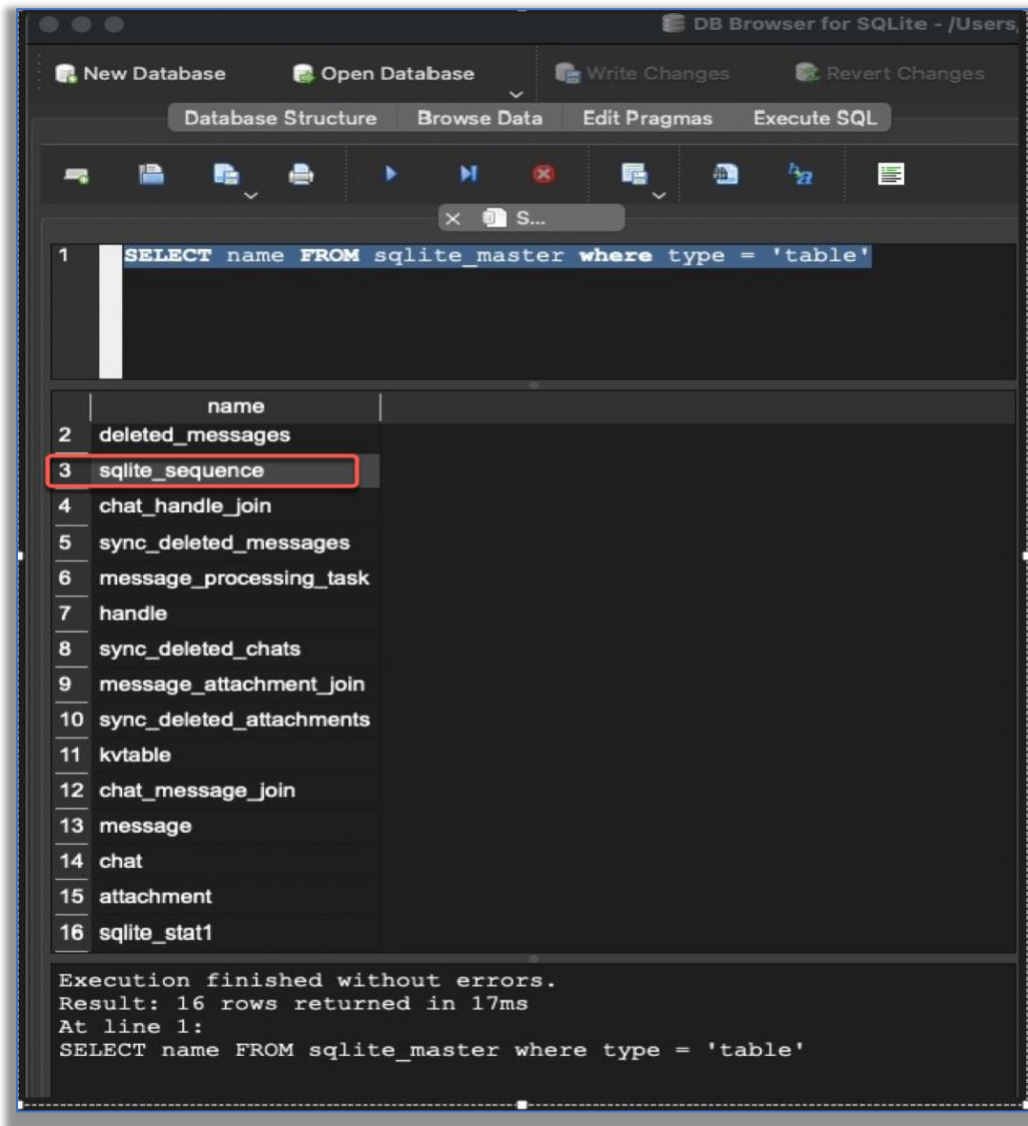


Figure 12: Screenshot from [DB Browser for SQLite](#) v 3.12.1 on macOS 11.0.1 showing search for all tables query.

With a slight variation to the previously shown query, the list of tables can be alphabetically ordered by name where the 'ORDER BY name' has been added.

```
SELECT * FROM sqlite_master WHERE TYPE = 'table' ORDER BY name
```

If you skip the first aforementioned approach, you can presume the internal `sqlite_sequence` table exists, and try to directly call the `sqlite_sequence` table using the next query. This will return a list of all the tables with their respective sequence (`seq`) values that are currently in use.

```
SELECT * from sqlite_sequence
```

Using the 'sms.db' SQLite database file (from Cellebrite CTF of Ruth Langmore's Apple iPhone X) with its associated WAL file, the screenshot below shows SQL query being used in DB Browser for SQLite v 3.12.1 on macOS 11.0.1. With successful execution of the query, you should now see all the tables in your database. Looking through the tables, it should be obvious which table you are interested in.

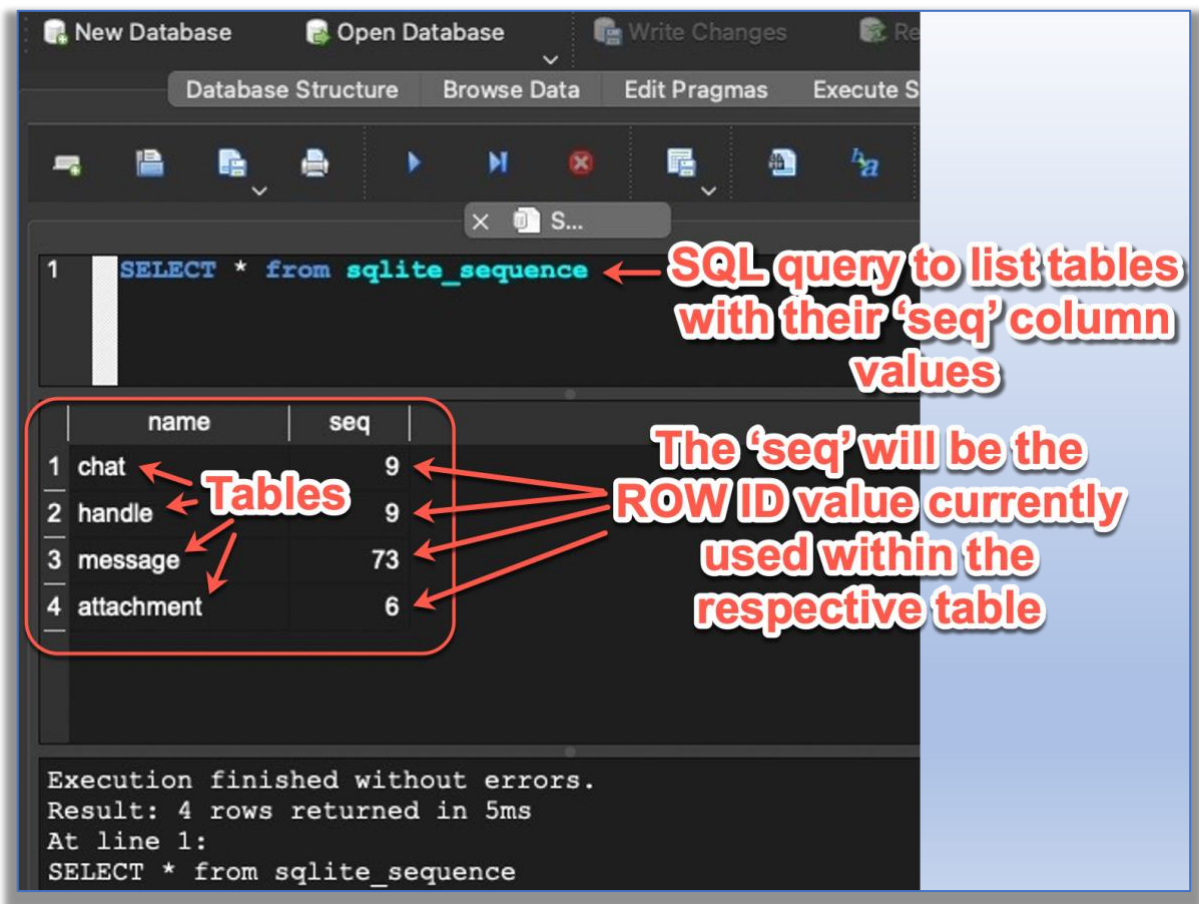


Figure 13: Screenshot from [DB Browser for SQLite](#) v 3.12.1 on macOS 11.0.1 showing "sqlite_sequence" query.

The "name" column shows you the names of tables that have automatically incrementing fields. The "seq" column shows you the latest id of that field. In the "messages" table, this means that the latest in-use sequence record number is 73. Thus, if you were to add a new row into the messages table, it would be assigned record number 74 and the "seq" field would be updated in the `sqlite_sequence` table to reflect the new record number.

You may ask what if there is no “sqlite_sequence” table. The “sqlite_sequence” table exists to handle the task of counting records. By using the AUTOINCREMENT field, the developer can offload the counting process to the SQLite database. The database will increment the record id sequentially. If a field with AUTOINCREMENT exists, then SQLite will not allow you to drop the “sqlite_sequence” table. If you do not find an “sqlite_sequence” table, then chances are the app isn’t using an AUTOINCREMENT field. In this case, it is incumbent upon the developer to handle the record counting.

The ‘CallHistory.storedata’ SQLite database, (from Cellebrite CTF of Ruth Langmore’s Apple iPhone X) is a great example of where there is no “sqlite_sequence” table, that is used to identify the current record in use within the tables present in the database. Instead, what is being used is the Z_PRIMARYKEY table that performs this function. The next screenshot shows the custom query search for the tables in the database. There are a total of 7 tables present, none of which are the “sqlite_sequence” table.

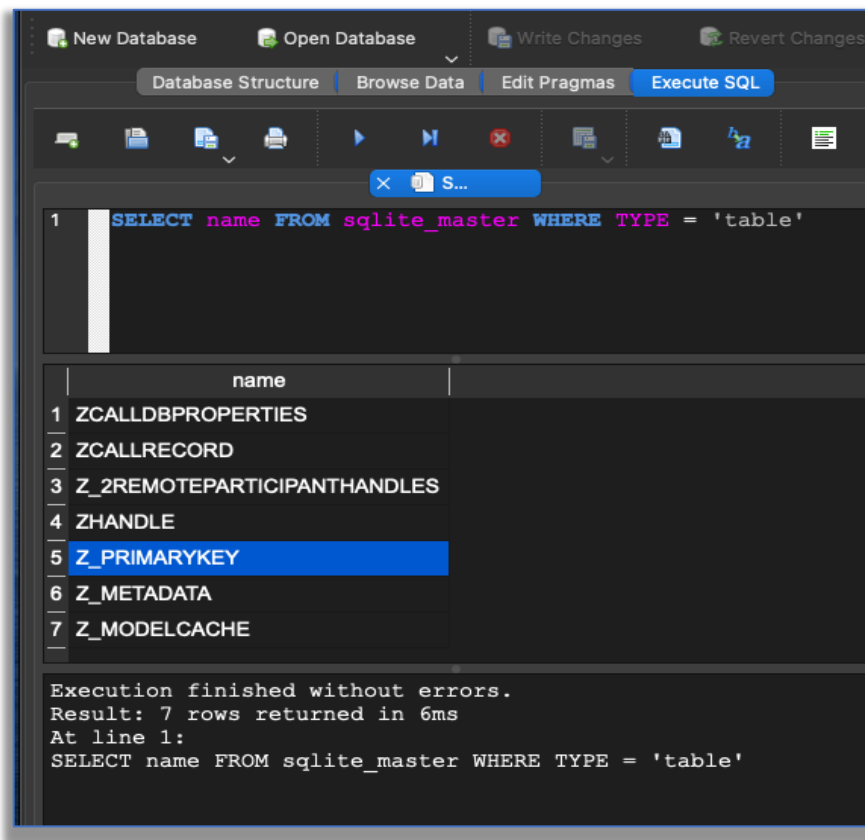


Figure 14: Screenshot from [DB Browser for SQLite](#) v 3.12.1 on macOS 11.0.1 showing the Z_PRIMARYKEY table listed, from Cellebrite CTF of Ruth Langmore’s Apple iPhone X.

There is however the Z_PRIMARYKEY table, as shown in the screenshot below, which provides essentially the same information as the “sqlite_sequence” table. The Z_MAX column in this case is used to identify the current record number in use within each of the tables in the ‘CallHistory.storedata’ SQLite database.

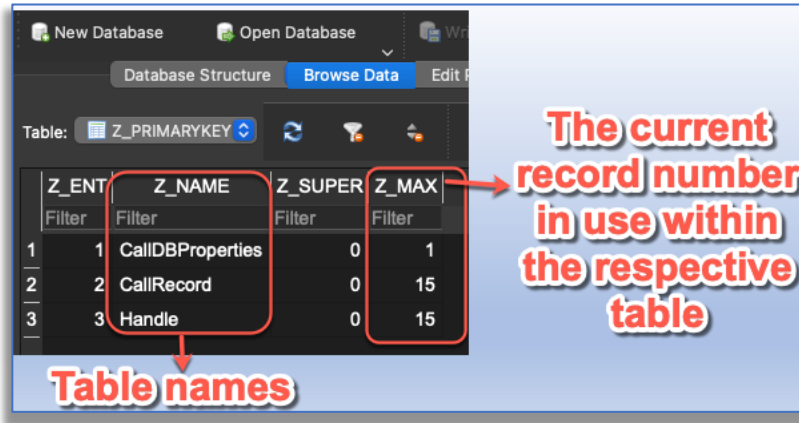


Figure 15: Screenshot from [DB Browser for SQLite](#) v 3.12.1 on macOS 11.0.1 showing the structure of the Z_PRIMARYKEY table, from Cellebrite CTF of Ruth Langmore’s Apple iPhone X

The ‘CallRecord’ table, under the Z_NAME column, from the previous screenshot, shows a Z_MAX value of 15. In comparison to the next screenshot of the ZCALLRECORD table, we can see that there are indeed 15 records that have been created thus far in this table, **but only 14 rows of records are displayed.**

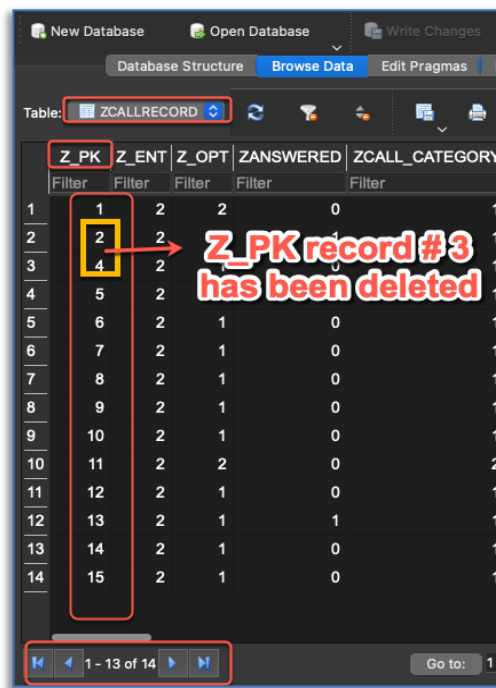


Figure 16: Screenshot from [DB Browser for SQLite](#) v 3.12.1 on macOS 11.0.1 showing the structure of the ZCALLRECORD table, from Cellebrite CTF of Ruth Langmore’s Apple iPhone X.

Don't focus on the row count but rather on the Z_PK column value, which is the record number (and is an alias for the ROWID column), which in this case identifies each unique record. There are 14 rows of records because the Z_PK column of the ZCALLRECORD table, has a gap between record 2 and record 4. Record 3 has been deleted! The use of the 'Z_' prefix column and table names appears to be unique to the Apple ecosystem itself. So, you may not see Z_' prefix column and table names in non-Apple SQLite databases.

Quick tip about the PRIMARY KEY table, cited directly from <https://www.sqlitetutorial.net/sqlite-primary-key/>: *"If a table has the primary key that consists of one column, and that column is defined as INTEGER then this primary key column becomes an alias for the rowid column."*

We have now learned how to identify a list of tables, and the current 'seq' (sequence) record value or PRIMARY KEY (PK) record number, that is in use, for tables within an SQLite database. Remember that a key principle in the operation of SQLite databases is the record value increments by one for every record added to the table, starting at record 1. Once a record number is deleted that record number value is never reused within an SQLite database. So, a table with a record value of 100 would be expected to have 100 records. This means that if there are only 99 records, and 1 through 99 are contiguous, then it must be record 100 that is missing.

This came in useful for Ian on a case where the user had deleted the incriminating photograph, which just so happened to be the last photograph taken. Using the 'seq' (sequence) record value, Ian was able to identify that there was a missing photograph, and due to the incremental way that iOS names photographs, he could work the name of the missing photograph itself. This led to finding the image as an email attachment that he was able to recover.

Missing Record Analysis

In this section we are going to be actually analyzing the missing records that exist within the 'CallHistory.storeddata' SQLite database file from [Josh Hickman's iOS 13.4.1 Public Image](#). This is a full file system extraction of an iPhone device, and therefore does contain the associated WAL file.

A quick check of the Z_PRIMARYKEY table, shows the 'CallRecord' table is currently using record 65, as identified in the Z_MAX column. We should, therefore, expect to see a record with a value of 65 in the Z_PK column of the ZCALLRECORD table, which we don't. Instead, we see a value of 64.

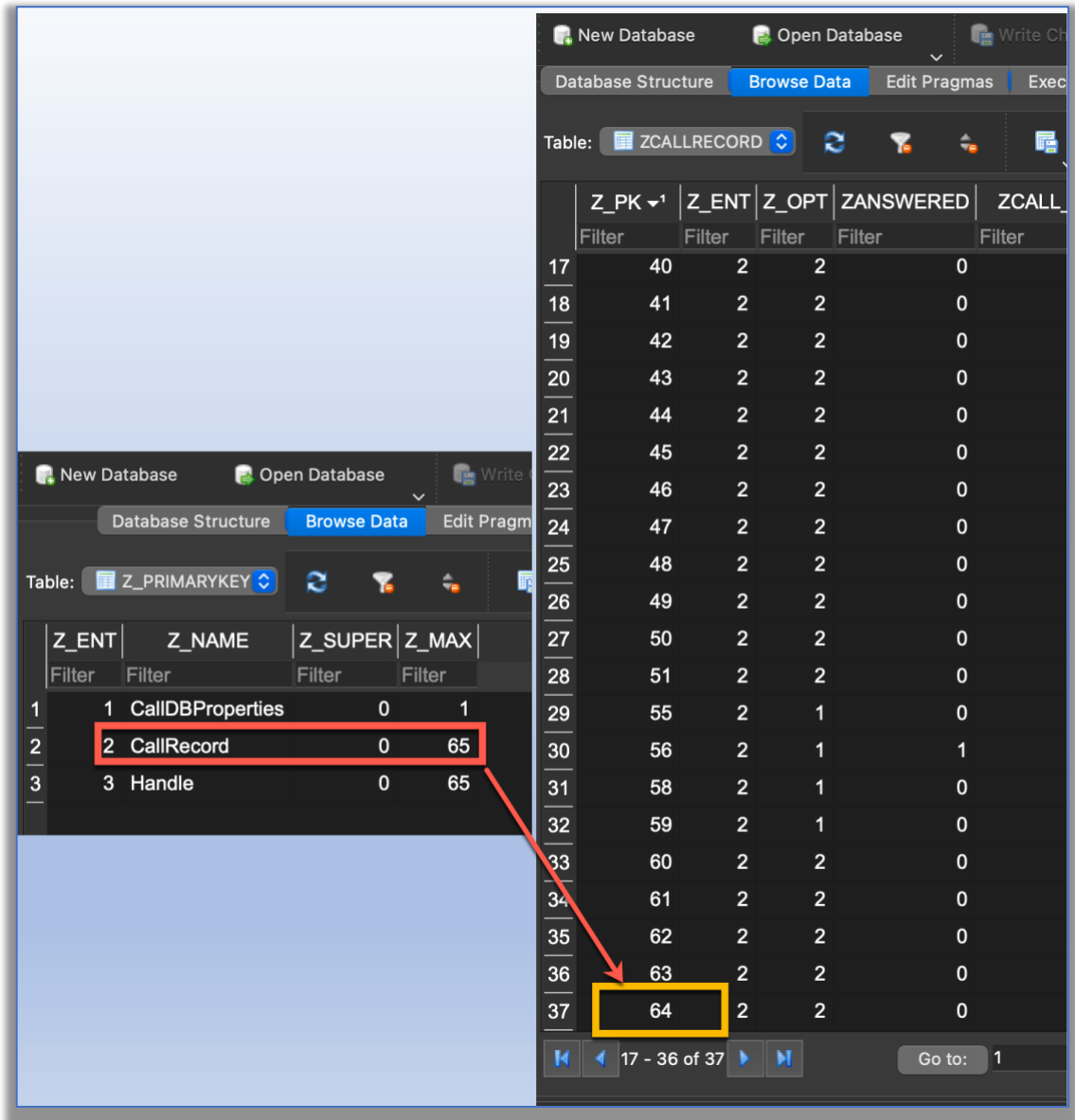


Figure 17: Screenshot from [DB Browser for SQLite](#) v 3.12.1 on macOS 11.0.1 showing the last record number 64 with record value 65 missing from the ZCALLRECORD table, from Josh Hickman's iOS 13.4.1 Public Image.

You should also have observed, from the above screenshot the evident disparity between the row number 37 and record number 64. This is a clear indication of the missing records within the ZCALLRECORD table of the 'CallHistory.storeddata' SQLite database file.

Missing Record Finder

It is possible to automate the analysis of missing records from an SQLite database. The next several screenshots show an example of a Windows based tool called Missing Record Finder developed by Ian Whiffin. He developed it when we worked together at our former law enforcement agency's digital forensics lab. At this time, Missing Record Finder is not publicly available.

Missing Record Finder v 1.7 simply checks for consecutive numbers in the relevant supported SQLite database tables. It does not attempt data recovery or comparisons. It also does not take the into account any unmerged live records from the WAL (or journal) file.

In this screenshot below, the left aspect of shows the Missing Record Finder 1.7, that has identified (without the WAL file) 35 existing records in the ZCALLRECORD table of the 'CallHistory.storedata' SQLite database file. The right aspect of the screenshot shows the same database opened with WAL file present, displaying the ZCALLRECORD table, in DB Browser for SQLite, presenting 37 existing records.

The screenshot is split into two panels. The left panel shows the 'Missing Record Identifier 1.7' application. It displays search criteria: Database 'E:\JH_CallHistoryDB - Copy\CallHistory.storedata', Date range from 2020/03/23 14:02:52 to 2020/04/14 09:56:12, and Database Type 'iOS Call History'. A note states: 'NOTE: This tool simply checks for consecutive numbers in the relevant db tables. It does not attempt data recovery or comparisons.' The search parameters are Start Date: 2020-03-23 2:02 PM, End Date: 2020-04-14 9:56 AM, Starting ID: 12, End ID: 62. A red box highlights the text: 'There SHOULD be 51 Records But only 35 were found (ie. 16 records are missing)'. A red arrow points to a red-bordered table with 35 rows. The first row is ID 12, TimeStamp 2020/03/23 14:02:52 (UTC-6), 3rd Party com.apple.Telephony, Direction Incoming. Rows 13-25 are red and labeled 'Missing'. Rows 26-28 are also present. A red-bordered table on the right shows 37 records with columns Z_PK, Z_ENT, Z_OPT, ZANSWERED. The first row is 1, 12, 2, 2. The last row is 37, 37, 2, 2. A red arrow points to the bottom of this table where it says '1 - 20 of 37'.

ID	TimeStamp	3rd Party	Direction
12	2020/03/23 14:02:52 (UTC-6)	com.apple.Telephony	Incoming
13	Missing		
14	Missing		
15	Missing		
16	2020/03/24 11:37:18 (UTC-6)	com.apple.Telephony	Incoming
17	Missing		
18	Missing		
19	Missing		
20	Missing		
21	Missing		
22	Missing		
23	Missing		
24	Missing		
25	Missing		
26	2020/03/26 11:51:45 (UTC-6)	com.apple.Telephony	Incoming
27	2020/03/27 10:25:36 (UTC-6)	com.apple.Telephony	Incoming
28	2020/03/27 13:55:03 (UTC-6)	com.apple.Telephony	Incoming

Z_PK	Z_ENT	Z_OPT	ZANSWERED
1	12	2	2
2	16	2	2
3	26	2	2
4	27	2	2
5	28	2	2
6	29	2	2
7	30	2	2
8	31	2	1
9	32	2	2
10	33	2	2
11	34	2	2
12	35	2	2
13	36	2	2
14	37	2	2
15	38	2	2
16	39	2	2
17	40	2	2
18	41	2	2
19	42	2	2
20	43	2	2
21	44	2	2

Figure 18: Missing Record Finder 1.7 showing presence of missing records within the ZCALLRECORD table of the 'CallHistory.storedata' SQLite database file.

The visual representation of missing records in Missing Record Finder 1.7 is an invaluable aid in understanding how many records are missing, in between specific time periods relative to existing records. What Missing Record Finder 1.7 does not do is identify missing records prior to the oldest existing record, or the very last missing record after the most recent existing record.

Another nice feature in the Missing Record Finder 1.7 output is displaying the SQLite custom queries that were used and a summary of missing records between date ranges. The date ranges are determined from existing records.

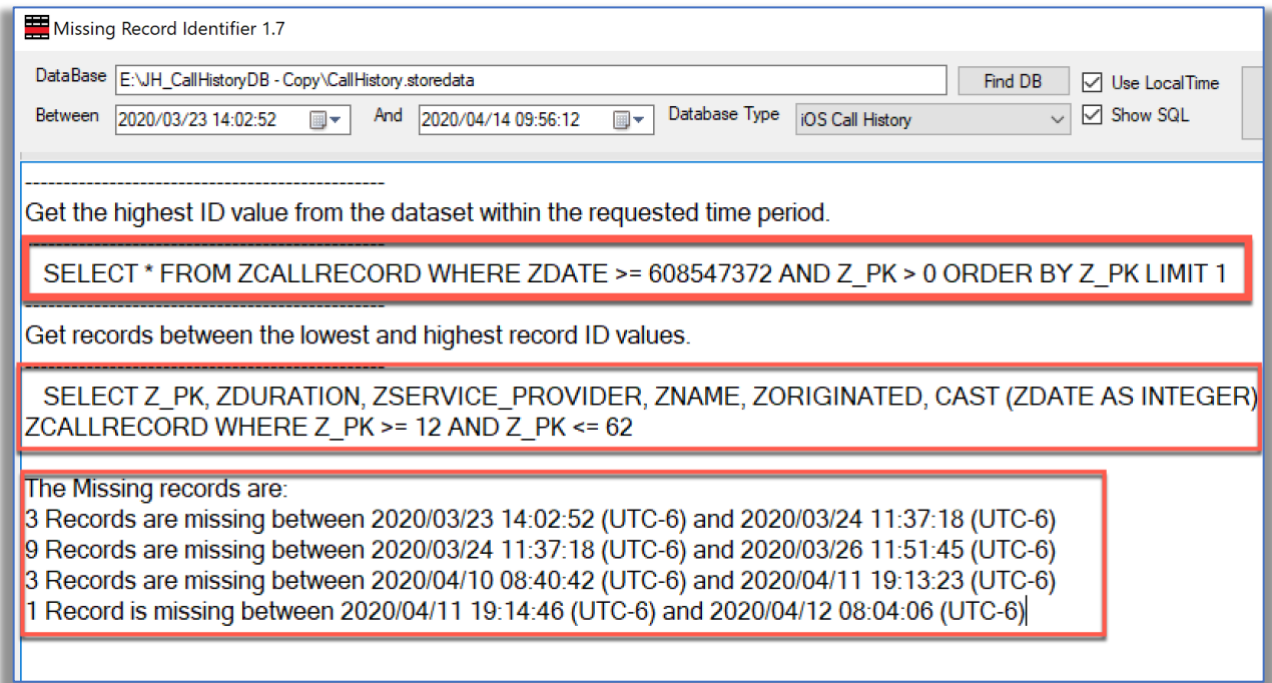



Figure 19: Missing Record Finder 1.7 showing a summary of missing records and the custom SQL query used to identify missing records in the ZCALLRECORD table of the 'CallHistory.storedata' SQLite database file.

ArtEx

Next, using [ArtEx](#) 1.5.1.0, a free open source iOS analysis tool (developed by Ian Whiffin), the 'CallHistory.storedata' SQLite database file was analyzed for missing records, from Josh Hickman's iOS 13.4.1 full file system (FFS) extraction. ArtEx is a Windows only tool, and was used, in this example, on Windows 10 20H2 (OS Build 19042.30) through VMFusion 12.1.0, running on macOS 11.0.1. It only accepts data from iOS full file system (FFS) extractions. It is not possible, to analyze a single SQLite database with ArtEx.

The following screenshot, in ArtEx, shows the existing call records parsed with identification of missing records, interjected in between existing records, based on the respective missing record number. In this view ArtEx currently does not show the missing records previous to the oldest existing record or after the most recent existing record. The record found count includes the existing and missing (not including the missing records previous to the oldest existing record or after the most recent existing record).



Click on any row under the 'Source Column' to open SQLite Viewer

Row ID	Icon	Start Time	End Time	Activity	Metadata	Source
1		2020-03-23 16:02:52 (UTC-05:00)	2020-03-23 16:02:52 (UTC-05:00)	Call (00:00:00)	Missed Audio Call from Unknown (+14082560700)	CallHistory.storedata (ZCALLRECORD Row 12)
2		2020-03-23 16:02:52 (UTC-05:00)	2020-03-24 13:37:17 (UTC-05:00)	3 Deleted Call Records		CallHistory.storedata (ZCALLRECORD Row 13 - 15)
3		2020-03-24 13:37:18 (UTC-05:00)	2020-03-24 13:37:18 (UTC-05:00)	Call (00:00:00)	Missed Audio Call from Unknown (+14082560700)	CallHistory.storedata (ZCALLRECORD Row 16)
4		2020-03-24 13:37:18 (UTC-05:00)	2020-03-26 13:51:44 (UTC-05:00)	9 Deleted Call Records		CallHistory.storedata (ZCALLRECORD Row 17 - 25)
5		2020-03-26 13:51:45 (UTC-05:00)	2020-03-26 13:51:45 (UTC-05:00)	Call (00:00:00)	Missed Audio Call from Unknown (+14082560700)	CallHistory.storedata (ZCALLRECORD Row 26)
6		2020-03-27 12:25:36 (UTC-05:00)	2020-03-27 12:25:36 (UTC-05:00)	Call (00:00:00)	Missed Audio Call from Unknown (+14082560700)	CallHistory.storedata (ZCALLRECORD Row 27)
7		2020-03-27 15:55:03 (UTC-05:00)	2020-03-27 15:55:03 (UTC-05:00)	Call (00:00:00)	Missed Audio Call from Unknown (+14082560700)	CallHistory.storedata (ZCALLRECORD Row 28)
8		2020-04-01 16:06:38 (UTC-05:00)	2020-04-01 16:06:38 (UTC-05:00)	Call (00:00:00)	Missed Audio Call from Unknown (+14082560700)	CallHistory.storedata (ZCALLRECORD Row 29)
9		2020-04-03 12:10:54 (UTC-05:00)	2020-04-03 12:10:54 (UTC-05:00)	Call (00:00:00)	Missed Audio Call from Unknown (+14082560700)	CallHistory.storedata (ZCALLRECORD Row 30)
10		2020-04-05 16:42:18 (UTC-05:00)	2020-04-05 16:42:41 (UTC-05:00)	Call (00:00:23)	Outgoing Audio Call to Unknown (9197627808)	CallHistory.storedata (ZCALLRECORD Row 31)
11		2020-04-06 16:34:33 (UTC-05:00)	2020-04-06 16:34:33 (UTC-05:00)	Call (00:00:00)	Missed Audio Call from Unknown (+14082560700)	CallHistory.storedata (ZCALLRECORD Row 32)
12		2020-04-06 17:48:08 (UTC-05:00)	2020-04-06 17:48:08 (UTC-05:00)	Call (00:00:00)	Missed Audio Call from Unknown (+14082560700)	CallHistory.storedata (ZCALLRECORD Row 33)
13		2020-04-06 18:43:00 (UTC-05:00)	2020-04-06 18:43:00 (UTC-05:00)	Call (00:00:00)	Missed Audio Call from Unknown (+14082560700)	CallHistory.storedata (ZCALLRECORD Row 34)
14		2020-04-07 14:10:42 (UTC-05:00)	2020-04-07 14:10:42 (UTC-05:00)	Call (00:00:00)	Missed Audio Call from Unknown (+14082560700)	CallHistory.storedata (ZCALLRECORD Row 35)

Figure 20: ArtEx 1.5.1.0 showing parsed existing and missing call records from 'CallHistory.storedata' SQLite database.

Within ArtEx, the SQLite viewer can be invoked either in two ways. First, through the Source column interface as shown in the screenshot above. If you prefer to analyze the SQLite database and its associated WAL file, then use the 'Directory View' method, as shown in the screenshot below.

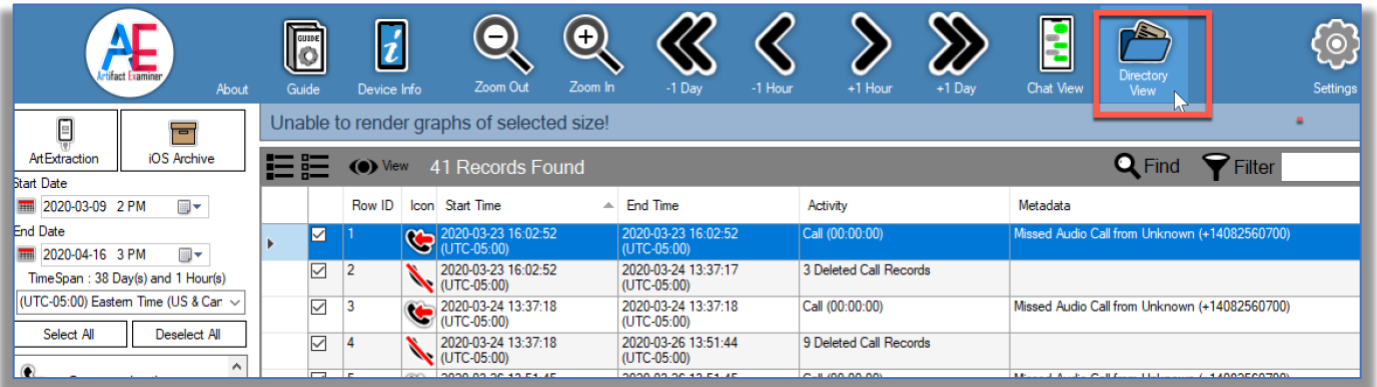


Figure 21: ArtEx 1.5.1.0 showing how to access Directory View.

Once 'Directory View' is opened, enter the database name and then click on 'Search Results' button. From the search results windows, click on the name of the SQLite database under the 'FileName' column to open the SQLite Viewer.

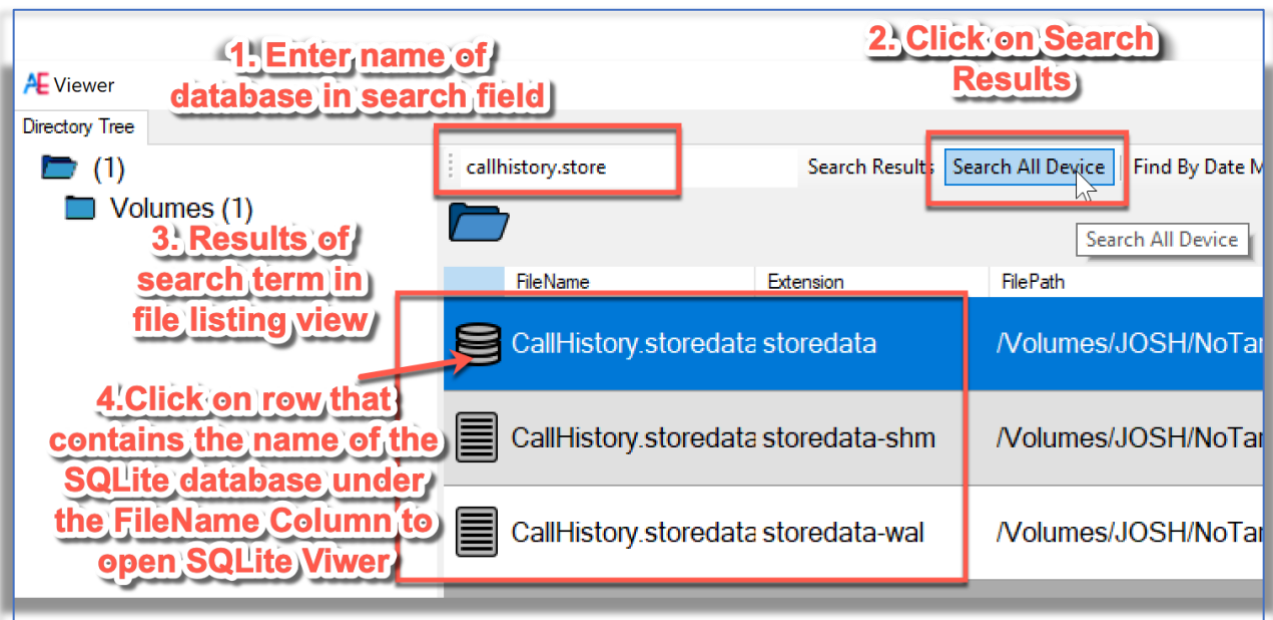


Figure 22: ArtEx 1.5.1.0 in Directory Tree view searching for CallHistory.storedata database.

The next screenshot shows the CallHistory.storedata SQLite database opened in SQLite Viewer (within ArtEx). There are 3 tabs from left to right. With WAL, Without WAL and WAL comparison. The missing records are not yet identified as that function must be initiated by the examiner. Make sure to first, select the database table you want to examine for missing records. Second, click on 'Find Missing Records'.

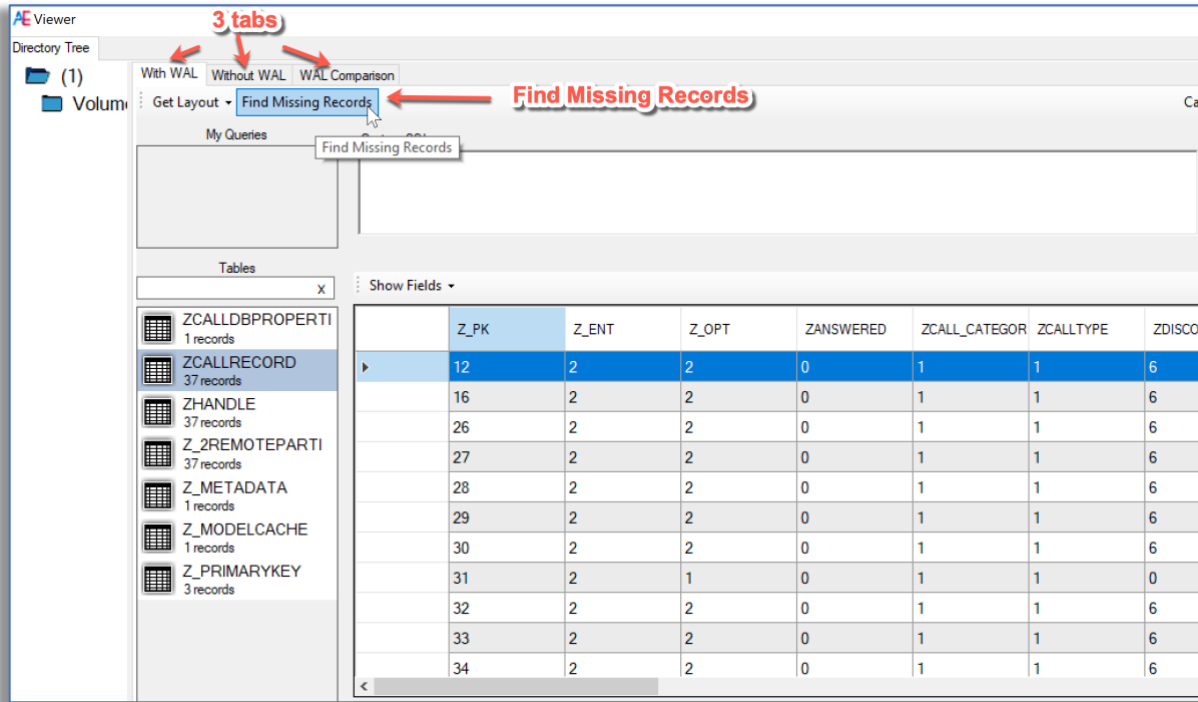


Figure 23: ArtEx 1.5.1.0 showing 'CallHistory.storedata' SQLite database open in SQLite Viewer, with 3 tabs: With WAL, Without WAL and WAL comparison.

After running the 'Find Missing Records' button, if missing records are identified, that are prior to the oldest existing record, in this case record 12, a message window will appear that identifies how many records were found to be missing but are not shown. The reason behind this is that if there are thousands of missing records prior to the oldest existing record, the digital forensics practitioner is made aware of it, as ArtEx only visually identifies missing records from the oldest existing record going forward to the most recent existing record.

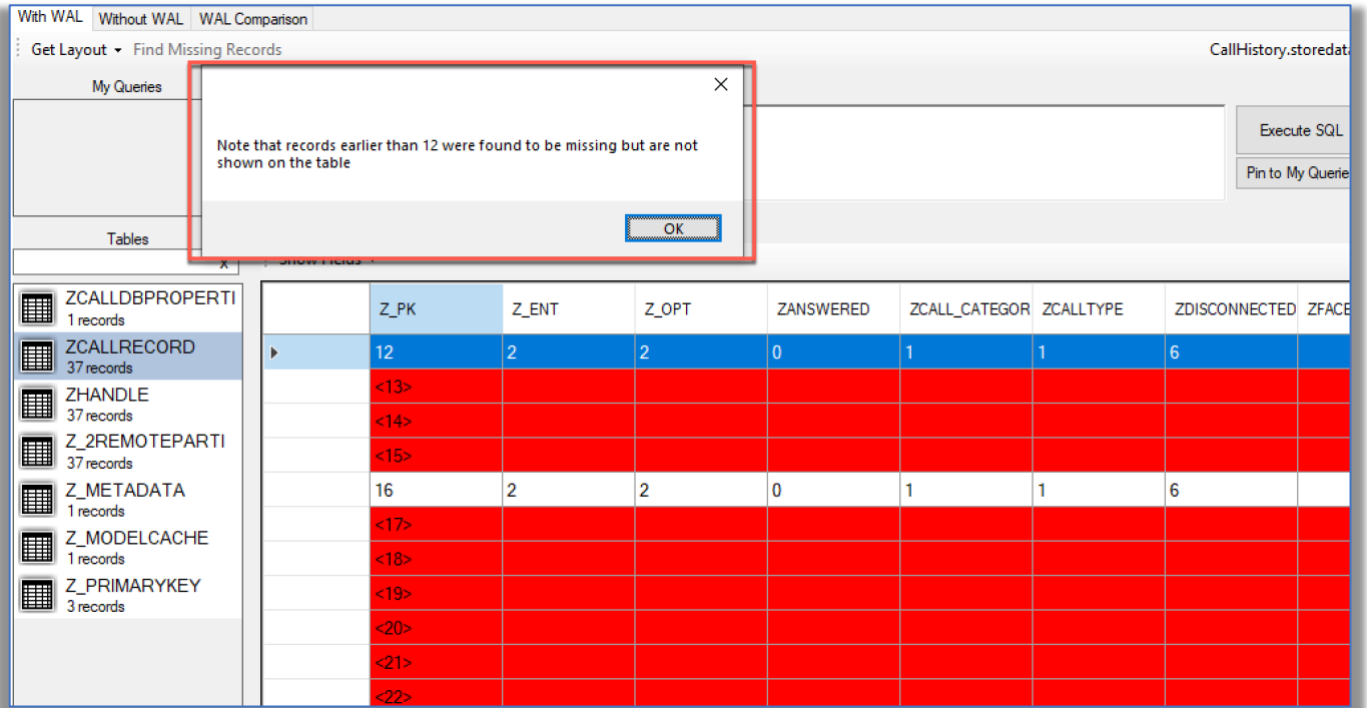


Figure 24: ArtEx 1.5.1.0, Find Missing Records function has been run on 'ZCALLRECORD' table ("CallHistory.storedata' SQLite database), WAL tab.

After the message windows (from the previous screenshot) has been acknowledged, the visual representation of missing records, highlighted as red coloured rows, can be observed.

The 'With WAL' tab shows the merged (added and removed) records between the WAL file and the database together, against all the tables within the database. This next screenshot shows 37 existing records, and 28 missing records found in the ZCALLRECORD table ('CallHistory.storedata' SQLite database). Missing record 65 is observed after the most recent existing record 64. (You can refer back to the screenshot in Figure 15 which shows this too, in DB Viewer for SQLite).

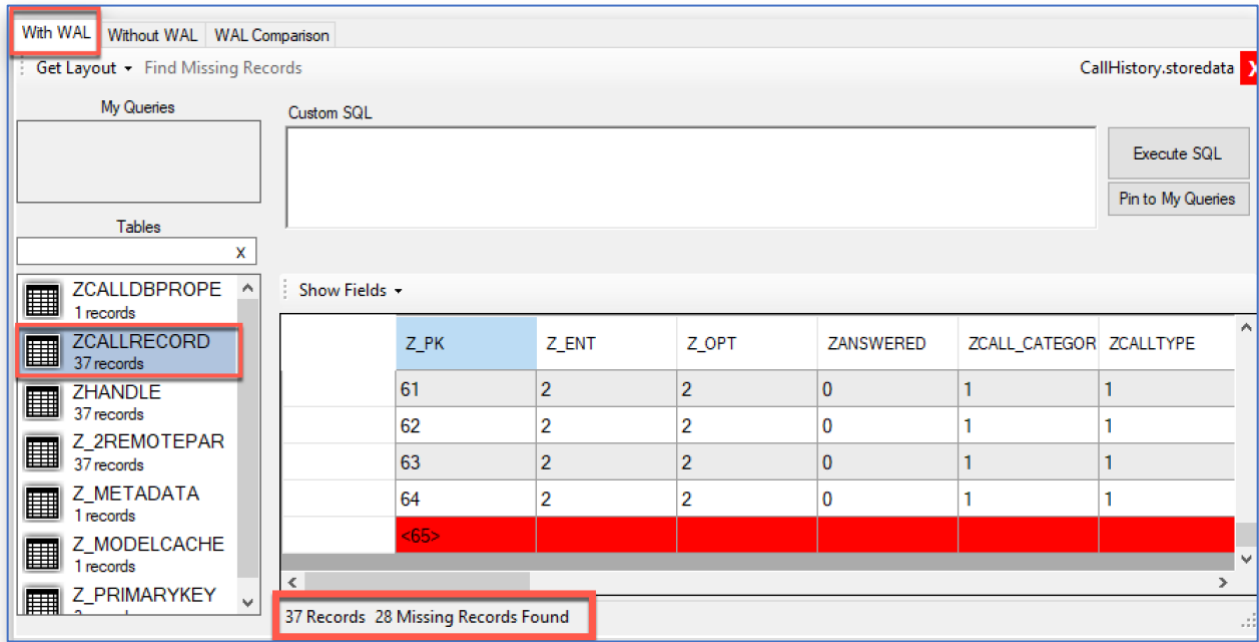


Figure 25: ArtEx 1.5.1.0, most recent record is identified as a missing record in the ZCALLRECORD table ('CallHistory.storedata' SQLite database), WAL tab.

The 'Without WAL' tab is the SQLite database only. The screenshot below, shows that the ZCALLRECORD table is an empty table, with zero (0) records. This means that the 'CallHistory.storedata' SQLite database file (without the WAL) contains no records! All the records have come from the WAL file. This allows you to infer the actual write (commit) operations to the SQLite database file have not yet taken place.

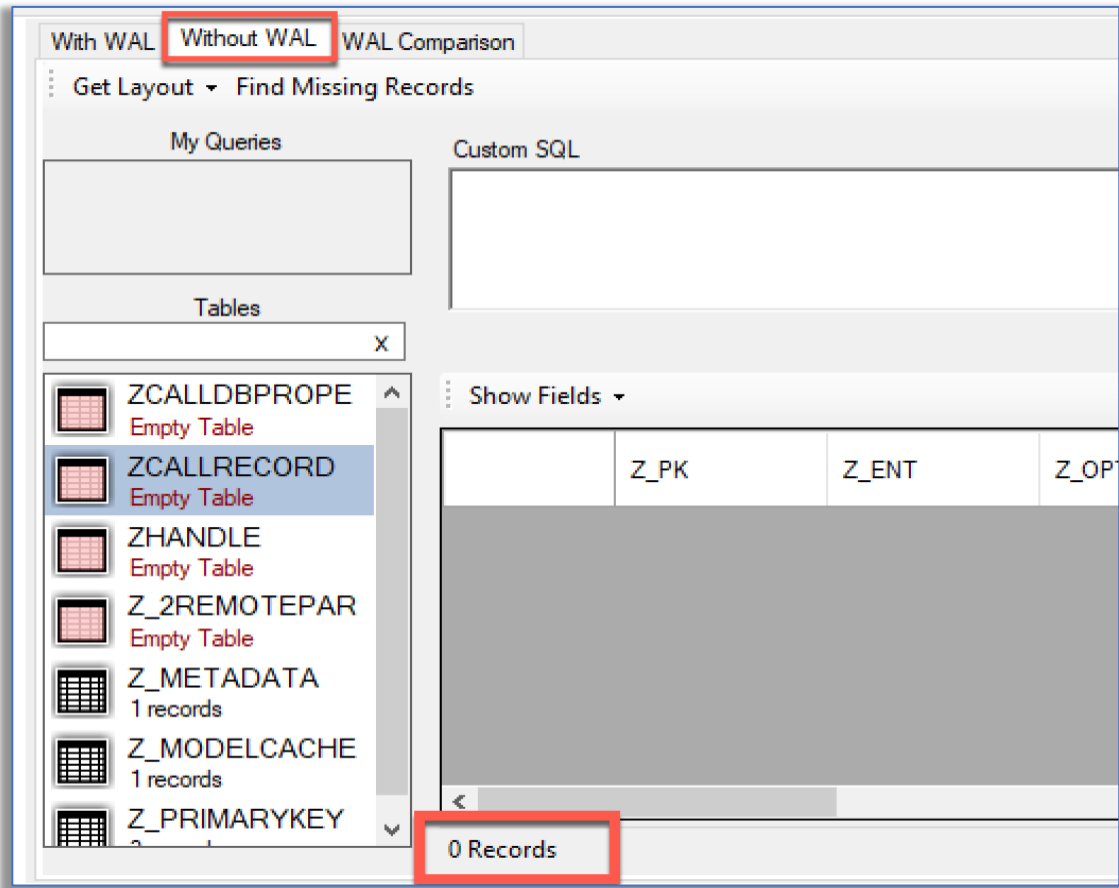


Figure 26: ArtEx 1.5.1.0, Find Missing Records function has been run on 'ZCALLRECORD' table ('CallHistory.storedata' SQLite database), Without WAL tab.

Trust but verify! The next screenshot shows the ZCALLRECORD table from 'CallHistory.storedata' SQLite database file (without the WAL), opened in DB Browser for SQLite. We just verified that there are no records present in the actual database file.

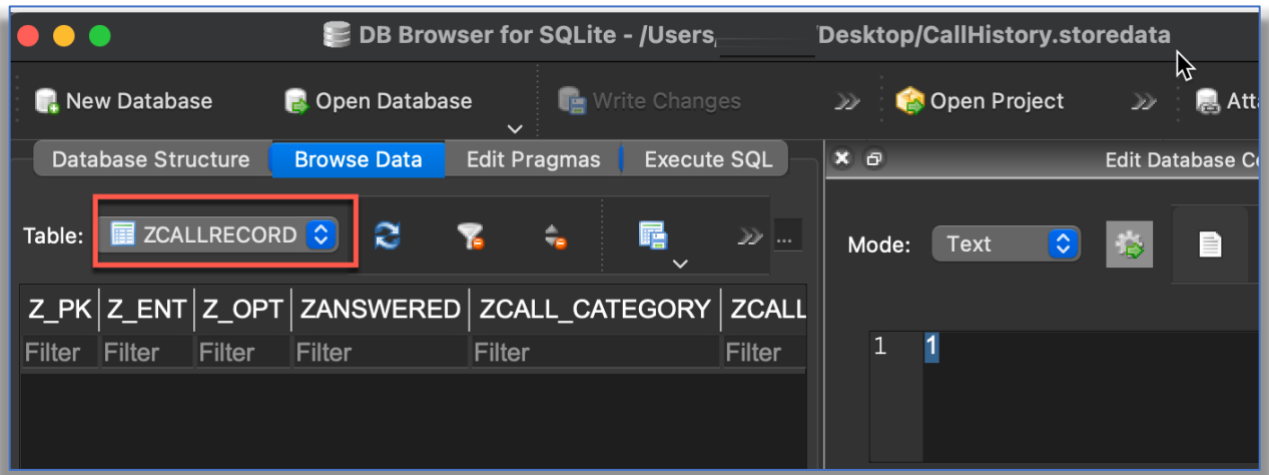


Figure 27: DB Browser for SQLite 3.12.1 showing no records in the ZCALLRECORD table from 'CallHistory.storedata' SQLite database file (without the WAL).

The 'WAL Comparison' tab provides a nice summary snapshot of all the tables in the SQLite database, being examined, and record number count with WAL and without WAL.

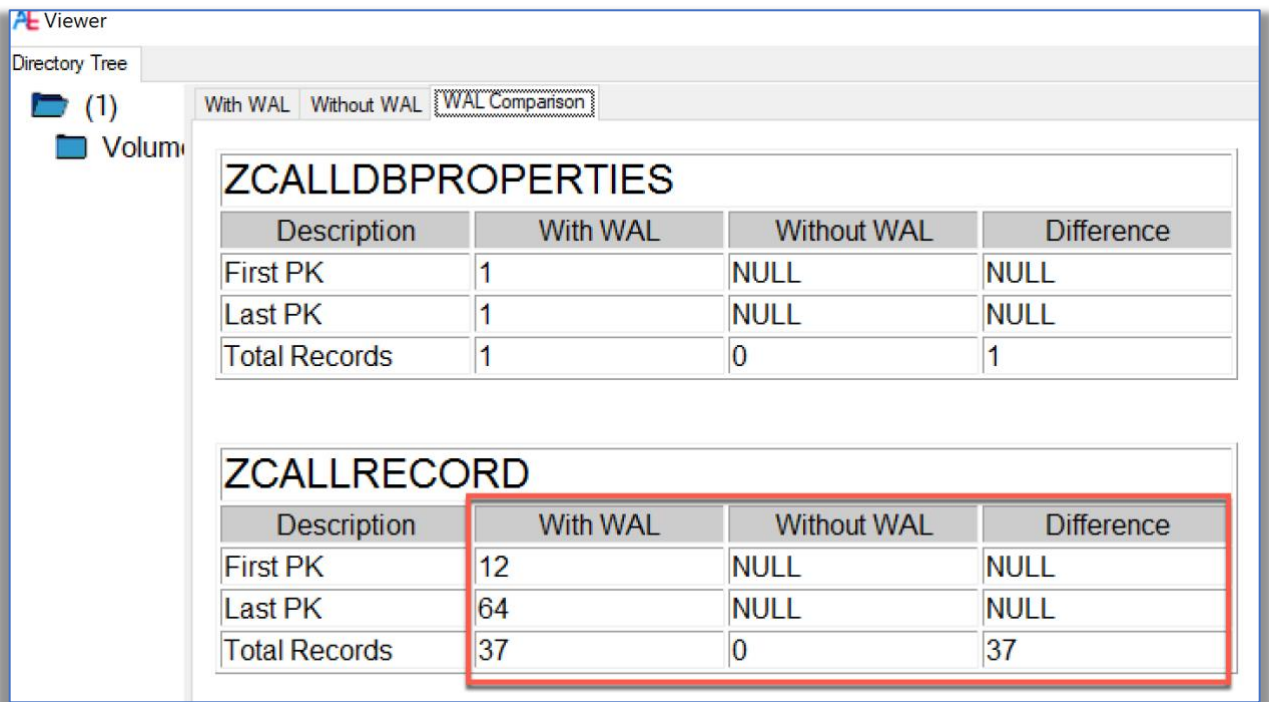


Figure 28: ArtEx 1.5.1.0 shows 'CallHistory.storedata' SQLite database, with identification of missing records

mirf (missing record finder)

This is an agnostic, free open source script, developed using Python 3, by Sheran Gunasekera which is available at <https://github.com/sheran/mirf>. The development of 'mirf' was inspired by the missing record analysis features in ArtEx. In its current iteration, 'mirf' is only designed to parse the 'CallHistory.storeddata' and 'sms.db' SQLite database files that are exported from iOS full file system extractions. It uses a predefined template to understand the structure of these two aforementioned SQLite database files.

The following screenshot of the Terminal app (on macOS 11.1) shows Python 3.9.0 (invoked through a pyenv environment) executing the 'mirf.py' script, to analyse a sms.db SQLite database file (exported from Josh Hickman's iOS 13.4.1 full file system (FFS) extraction) for missing records. The analysis results are straight forward and easy to interpret. In this case 2 missing records are identified, including date ranges in UTC, and the missing record number.

```
I [mirf:3.9.0] ~/c/mirf · main ★
)) python mirf.py JH_SMS/sms.db
[i] Parsing: iOS SMS DB file 'message' table
[i] First Record ID:                               1
[i] Last Record ID:                               48
[i] Last Record ID in message table according to sqlite_sequence table: 48
[i] Total Records:                                 46
[i] Missing Record Count:                          2

[i] Missing Records List:

1. 1 record(s) missing between Wed Mar 25 00:49:24 2020 (UTC) and Thu Mar 26 01:11:36 2020 (UTC).
   Missing record numbers are: [12]

2. 1 record(s) missing between Mon Mar 30 08:15:36 2020 (UTC) and Wed Apr 1 17:39:22 2020 (UTC).
   Missing record numbers are: [18]
```

Figure 29: mirf.py analyzing sms.db (exported from Josh Hickman's iOS 13.4.1 full file system (FFS) extraction) which contains 2 missing records.

The next screenshot shows the 'mirf.py' script being run (in the same environment as previously noted), analysing the 'CallHistory.storedata' SQLite database files (exported from Josh Hickman's iOS 13.4.1 full file system (FFS) extraction) for missing records. The analysis of this database shows that there is a total of 28 missing records. The Missing Records List shows the details of all missing records. For example, that records 1 through 11 are missing from before 20:02:52 on 23rd March 2020 (UTC). Also note that in relation to the Last Record ID value that was used for ZCALLRECORD table is 65, relative to the Last Record ID value of 64 that is present. This means the last record in the ZCALLRECORD table is missing.

```
))) python mirf.py JH_CallHistoryDB/CallHistory.storedata
[i] Parsing: iOS CallHistory DB file 'ZCALLRECORD' table
[i] First Record ID: 12
[i] Last Record ID: 64
[i] Last Record ID in ZCALLRECORD table according to Z_PRIMARYKEY table: 65
[i] Total Records: 37
[i] Missing Record Count: 28

[i] Missing Records List:

1. 11 deleted record(s) before Mon Mar 23 20:02:52 2020 (UTC).
   Missing records are: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

2. 3 deleted record(s) between Mon Mar 23 20:02:52 2020 (UTC) and Tue Mar 24 17:37:18 2020 (UTC).
   Missing records are: [13, 14, 15]

3. 9 deleted record(s) between Tue Mar 24 17:37:18 2020 (UTC) and Thu Mar 26 17:51:45 2020 (UTC).
   Missing records are: [17, 18, 19, 20, 21, 22, 23, 24, 25]

4. 3 deleted record(s) between Fri Apr 10 14:40:42 2020 (UTC) and Sun Apr 12 01:13:23 2020 (UTC).
   Missing records are: [52, 53, 54]

5. 1 deleted record(s) between Sun Apr 12 01:14:46 2020 (UTC) and Sun Apr 12 14:04:06 2020 (UTC).
   Missing records are: [57]

6. 1 deleted record(s) after Tue Apr 14 15:56:12 2020 (UTC).
   Missing records are: [65]
```

Figure 30: mirf.py analyzing 'CallHistory.storedata' (exported from Josh Hickman's iOS 13.4.1 full file system (FFS) extraction) which contains 28 missing records.

It is anticipated that further development of 'mirf.py' will take place to allow for a guided (versus template) mode of analysis of missing records from any SQLite database. This method of analysis would rely on the digital forensics practitioner, based on prompts from analysis tool, to identify the relevant tables(s) and their respective fields to be analysed.

Final Thoughts

The purpose behind the reader having to understand the pages of a sometimes dry and somewhat challenging concept of SQLite databases, is to make the digital forensics practitioner aware of the value of missing records analysis.

After processing and understanding the concepts explained in the previous pages, there are **generalized inferences** about missing records, that can be explained as follows:

1. An identified missing record, that is after the most recent existing record, indicates with a high degree of professional certainty that the device user has intentionally deleted a record.
2. Equally, a group of missing records that exist between a set of two existing records, indicates with a high degree of professional certainty that the device user has intentionally deleted these records.
3. If timestamps are present for existing records, this provides a temporal context relative to the existing records, of when the missing/deleted records were originally created.
4. The exact date and time, a missing record was deleted cannot be ascertained.
5. Who performed the action of deleting records cannot be ascertained from the identification of missing records alone.
6. If a database table is empty and missing records are identified, in conjunction with the presence of the most recent record value in use, this would indicate with a high degree of professional certainty that the device user has intentionally deleted these records.

While the absence of a record number signifies deletion of a row, it is incumbent on the digital forensics practitioner to **distinguish between system deleted or user deleted records**. This can only really be done by determining if the user has access to the database in question. For example, a record from an SMS database would be deleted when the user deletes the message. But system databases aren't typically accessible to the user so deleted records here would be assumed to be system deleted. Furthermore, if possible, application settings should also be checked on the device and/or in the application structure (from the extracted data), in order to determine how long data is specifically retained by that application. These settings can greatly influence in determining whether the presence of missing records in an SQLite database are caused by the actions of the device user or by the device itself.

Finally, thank you for taking the to read this article and we hope you enjoyed it and learned concepts that you can use your digital forensics tradecraft!

Sources/References

1. SQLite: <https://www.sqlite.org/index.html>
2. SQLite Application File Format: https://www.sqlite.org/aff_short.html
3. SQLite Archive Files: <https://www.sqlite.org/sqlar.html>
4. wal and shm files : <https://www.sqlite.org/walformat.html>
5. Temporary Files Used By SQLite: <https://sqlite.org/tempfiles.html>
6. Write-Ahead Logging: <https://www.sqlite.org/wal.html>
7. Clustered Indexes and the WITHOUT ROWID Optimization:
<https://www.sqlite.org/withoutrowid.html>
8. Sanderson, P. (2018). *SQLite Forensics* (1st ed.). Paul Sanderson.

Page 268 Paraphrased: The ROWID value, is an integer primary key that is defined as autoincrement. This ensures that each new record added to the database will have a unique key that is one more than the previously used maximum value. The ROWID integer primary key values are therefore contiguous, with no gaps in the integer values.

9. Sanderson, P. (2018). *SQLite Forensics* (1st ed.). Paul Sanderson.

Page 69 Paraphrased: Implemented in SQLite version 3.3.5, 'secure_delete' status is not stored in the SQLite database; when investigating an SQLite database, there is nothing that a digital forensics practitioner can check to ascertain if secure delete is enabled. If an SQLite database is compiled with the 'secure_delete' option, then deleted records are overwritten with zeroes.

10. SQLite POCKET REFERENCE GUIDE (by Lee Crognale, Heather Mahalik and Sarah Edwards):
<https://digital-forensics.sans.org/media/SQLite-PocketReference-final.pdf>

11. The iPhone Data Recovery Myth: What You Can and Cannot Recover (July 10th, 2020 by Oleg Afonin): <https://blog.elcomsoft.com/2020/07/the-iphone-data-recovery-myth-what-you-can-and-cannot-recover/>

“Your text messages and iMessages are stored in a database in the SQLite format. By default, SQLite does not overwrite records immediately after they’ve been deleted. Instead, SQLite marks them as “deleted”. Deleted pages become unused and are stored on what is called a “freelist”. If you obtain the database files (by making a backup), these records can be recovered until the moment the database is fully vacuumed and defragmented (if it is, the deletion becomes permanent). This used to be the case in iOS 8 through iOS 11. Starting with iOS 12, Apple seemingly moved to a non-standard implementation, physically wiping records almost immediately after they are deleted. As a result, deleted text messages and iMessages cannot be recovered in iOS 12, 13 and newer.”

“The one problem of recovering deleted records (be it messages, call logs or contacts) is the volatile nature of SQLite databases in modern versions of iOS. The only easy way to obtain SQLite databases from the device is making an iTunes backup. Until you make the backup, the databases are used with unmerged WAL (write-ahead logs), and However, the very moment you initiate the backup, the SQLite databases are merged, and the deleted records are lost forever.”

12. SQLite Autoincrement: <https://sqlite.org/autoinc.html>
13. ArtEx by Ian Whiffin (registration required): <https://www.doubleblak.com/>
14. Josh Hickman’s iOS 13.4.1 Public Image: <https://thebinaryhick.blog/?s=13.4.1>
15. Missing Record Finder Python 3 script, mirf.py: <https://github.com/sheran/mirf>

Acknowledgements

The authors would like to thank Sheran Gunasekera for taking the time to review this article and providing guidance. Over 10 years ago Sheran and Shafik collaborated extensively on BlackBerry Forensics research and recovery of SQLite deleted records.

Sheran is also the author of two Android App Security books:

1. Blue Team perspective published in September 2012.
Android Apps Security Blue:
<https://www.amazon.ca/Android-Apps-Security-Sheran-Gunasekera/dp/1430240628>
2. Red Team perspective published November 2020:
Android Apps Security: Mitigate Hacking Attacks and Security Breaches
<https://www.amazon.ca/Learn-Android-Security-Stack-Zhauniarovich/dp/1484216814>