Forensics from the sausage factory

I worked at the coal face of a UK computer forensics lab and performed production line forensics - day in day out - welcome to the sausage factory

Wednesday, 4 May 2011

SQLite Pointer Maps pages

This blog post complements and should be read in conjunction with the previous post Carving SQLite databases from unallocated clusters. In that post I looked at the information available within an SQLite database that may assist in carving one from unallocated clusters.

You will remember from the earlier blog post that SQLite databases are divided into equally sized *pages and* SQLite database files always consist of an exact number of *pages.* The *page* size for a database file is determined by the 2 byte integer located at an offset of 16 bytes from the beginning of the database file. The first *page* in an SQLite database is numbered *page* 1.

Auto-vacuum capable

Auto-vacuum capable SQLite databases make use of Pointer Map *pages* along with the other *page* types detailed in the earlier blog post. It is probably helpful to provide some information about what an auto-vacuum capable database is.

In a non-auto-vacuum-capable SQLite database when information is deleted the *pages* where it was stored are added to a list of free *pages* and these pages can be reused the next time data is inserted. Therefore, should data be deleted the file size of the database does not decrease. If a lot of data is deleted and it becomes necessary to shrink the database size the SQL VACUUM command can be run. This has the effect of reorganising the database from scratch and removing any free pages completely, thus making the database smaller.

When Auto-vacuum is enabled all free *pages* are moved to the end of the database file and the database file is truncated to remove the free *pages* at every transaction commit, thus removing free *pages* automatically.

Pointer Map Pages

The purpose of the Pointer Map is to facilitate moving pages from one position in the database file to another as part of auto vacuum. When a page is moved, the pointer in its parent must be updated to point to the new location. Pointer Maps are used to provide a lookup table to quickly determine what a

Where to find me

No longer at the sausage factory but you may find me on my bike looking for work!

My name is Richard Drinkwater. I am a freelance computer forensics consultant and welcome enquiries sent to DC1743 (at) gmail dot com

LinkedIn profile

Factory Clock

Please Subscribe To This B

Posts

Comments

pages parent *page* is. They only exist within auto-vacuum capable databases, which require the 32 bit integer value, read big endian, stored at byte offset 52 of the database header to be non-zero.

In auto-vacuum-capable SQLite databases *page* 2 of the database is always a Pointer Map *page.* Pointer Map *pages* store a 5 byte record relating to every page that **follows** the Pointer Map *page. For example* if we have an auto-vacuum-capable database that has **24 *pages*** (each of 4096 bytes in size) in total, *page* 1 will contain the database header and the database schema and the next *page, page* 2, will be a Pointer Map *page.* This Pointer Map *page* will contain a 5 byte record for **every one** of the remaining **22 *pages*** taking up 110 bytes of space within the *page*. The first 5 byte record begins at the very beginning of the Pointer Map *page* and therefore in a 4096 byte *page* a maximum of 819 (4096/5) records can be stored. If the database has more than 821 *pages* (when using a *page* size of 4096 bytes) *page* 822 would be an additional Pointer Map page that would contain records for the next 819 *pages* **following** this second Pointer Map *page*. Further additional Pointer Map *pages* can be added in the same way. Pointer Map *pages* do not store records relating to Pointer Map *pages* or *page* 1 of the database.

Pointer Map 5 byte records are structured with 1 byte indicating a Page Type and then 4 bytes, decoded big endian, referencing the parent page number as follows:

- **0x01** 0x00 0x00 0x00 0x00
  This record relates to a B-tree root page which obviously does not have a parent page, hence the page number being indicated as zero.
- **0x02** 0x00 0x00 0x00 0x00
  This record relates to a free page, which also does not have a parent page.
- **0x03** 0xVV 0xVV 0xVV 0xVV (where VV indicates a variable)
  This record relates to the first page in an overflow chain. The parent page number is the number of the B-Tree page containing the B-Tree cell to which the overflow chain belongs.
- **0x04** 0xVV 0xVV 0xVV 0xVV (where VV indicates a variable)
  This record relates to a page that is part of an overflow chain, but not the first page in that chain. The parent page number is the number of the previous page in the overflow chain linked-list.
- **0x05** 0xVV 0xVV 0xVV 0xVV (where VV indicates a variable)
  This record relates to a page that is part of a table or index B-Tree structure, and is not an overflow page or root page. The parent page number is the number of the page containing the parent tree node in the B-Tree structure.

Websites I go back to

Computer Forensics Resources

Digital Detective Forums

Forensicwiki

Macosxforensics

Slyck

Wotsit

Blogs I read

Leo Leporte

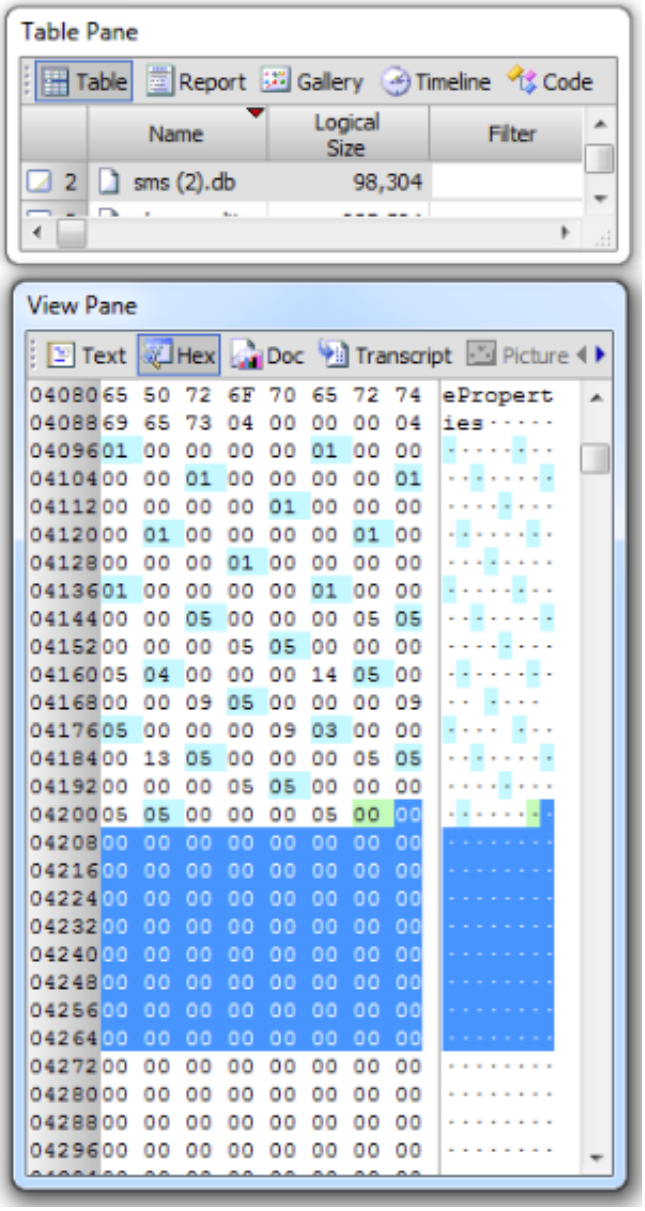

Figure 1

The screenshot at **Figure 1** shows the Pointer Map page of an iPhone SMS.db which uses a page size of 4096 bytes. The Page Type bytes are highlighted in light blue and occur at every fifth byte. The byte highlighted in green (0x00) is the first byte of the sequence that is not one of the page type bytes as described above and therefore indicates that there are no more records stored in this pointer map as can be seen within the highlighted darker blue area.

Extrapolating the database size from the Pointer Map page

If you count the Page Type bytes highlighted within **Figure 1** in light blue you will find there are 22. This is because we have 22 pages following the Pointer Map page and therefore require 22 records. This allows us to conclude that there are 24 pages in total within this database (page 1, the Pointer Map page and

Lance Mueller's blog
Harlan's blog
Happy as a monkey
Andre Ross's blog

Podcasts I listen to

Cyberspeak
MacBreak Weekly

Blog Archive

► 2013 (1)
► 2012 (4)
▼ 2011 (5)
► July (1)
▼ May (2)
An analysis of the record structure within SQLite ...
SQLite Pointer Maps pages
► April (1)
► January (1)
► 2010 (21)
► 2009 (25)
► 2008 (24)

then the 22 pages). By examining the 2 byte integer located at an offset of 16 bytes from the beginning of the database file we have determined that the page size within this database is 4096 bytes. 24 multiplied by 4096 equals 98304. The file size of this particular database is therefore 98,304 bytes which can also be seen within **Figure 1**.

Carving Considerations
To carve auto vacuum capable databases the following steps would be needed:

- Identify first page of database by detecting the *SQLite format 3* header
- Establish *page size* by reading the 2 bytes at offset 16 as a 16 bit integer big endian
- Check the 4 byte 32 bit integer at offset 52 for a non zero value indicating that the database is auto vacuum capable
- Go to page 2 of the database at Offset *page size*
- If value is 0x01 or 0x02 or 0x03 or 0x04 or 0x05 set a counter to 1
- Move five bytes forward and if value is 0x01 or 0x02 or 0x03 or 0x04 or 0x05 increment the counter by 1
- **or** if the value is not 0x01 or 0x02 or 0x03 or 0x04 or 0x05 begin the calculation of database file size using the formula

- *database size = (counter value + 2) x page size*

The above discounts the possibility of their being more than one pointer map *page*. Some additional logic would be needed to cater for this eventuality. Pointer map *pages m*ay contain *page size/5* records. If the counter increments to a point where it equals this value it would be necessary to locate the next pointer map *page* using the formula:

next pointer map page number = (page size/5) + 2 + number of existing pointer map pages.

To calculate the offset to this *page* use the formula:

offset = (page number-1) x page size.

References
http://www.sqlite.org/fileformat.html
http://www.sqlite.org/fileformat2.html
http://www.sqlite.org/src/artifact/cce1c3360c

Posted by DC1743 at 12:09

1 comment:

Anonymous said...

Great post. Thanks for taking the time to post this and the previous one. I just finished working a case involving an iPhone and this has helped explain things. I may actually go back and take another look at it.

-Steve

5 May 2011 18:09

Post a Comment

Links to this post

Create a Link

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)