SQLite Records Recovery

Parser di record rimossi

Introduzione

SQLiteRecordsRecovery è uno strumento per trovare e recuperare i records rimossi da database SQLite.

Lo script è scritto in python versione 3.8.10.

Lo strumento opera in due modalità che saranno eseguite entrambe ad ogni esecuzione.

La prima modalità si limiterà a prendere i dati delle aree in cui potrebbero essere presenti dei record e li immagazzinerà in un file chiamato “raw_data.tsv”.

La seconda modalità a partire dai dati ottenuti dalla modalità precedente cercherà di parsare i records o frammenti di record.

Terminata l’esecuzione sarà possibile visualizzare i record ottenuti tramite il file “report.html” (assicurarsi che il browser utilizzato per l’apertura del file html permetta il caricamento di file json locali) (per maggiori informazioni visitare la pagina GitHub)

Funzioni

Questo strumento consente di:

- recuperare records dallo spazio non allocato delle pagine di tipo leaf table B-tree

- recuperare records dai freeblocks presenti nelle pagine di tipo leaf table B-tree

- supporta le diverse codifiche utilizzate da SQLite per immagazzinare le stringe (UTF-8, UTF-16BE, UTF-16LE)

- salvare i bytes delle aree non allocate e dei freeblocks su un file tsv

- visualizzare i dati ottenuti tramite una interfaccia

Link utili

Github: https://github.com/Er-Simon/Sqlite3-restore-deleted-records

Formato dei database file di tipo SQLite: https://wwwsqlite.org/fileformat.html

Funzionamento

Nel file main.py la variabile database_path conterrà il path in cui è collocato il file del database.

Verrà creato un oggetto della classe database (database_parser.py) rappresentante il file situato al database_path, durante la creazione verificherà che il file esista, otterrà la dimensione in byte del file e verrà controllato che l'utente abbia i permessi di lettura.

Se le condizioni precedenti sono verificate procederà ad aprire il file in modalità "rb" (reading binary) e ne leggera i primi 100 byte (l'header del database).

Tramite l'header controllerà che i primi 16 byte convertiti in hex corrispondano alla stringa "53514c69746520666f726d6174203300", in quanto ogni SQLite database file valido inizia con i precedenti byte.

Leggerà i successivi 2 byte (offset 16) per acquisire la grandezza delle pagine in byte e successivamente all'offset 56 ne leggerà ulteriori 4 per ottenere un intero rappresentante la codifica utilizzata per immagazzinare le stringhe nel database.

Per ulteriori informazioni relative alla struttura dell'header utilizzare il seguente link:
https://www.sqlite.org/fileformat.html#the_database_header

Terminata la fase di inizializzazione verrà creato un oggetto della classe report presente nel file (report_builder.py). Questo oggetto rappresenta le informazioni ottenute durante l'esecuzione dello script, inoltre, al termine dell'esecuzione genererà un file "report.html" per visualizzare le informazioni ottenute.

Aperto il flusso di dati del file e ottenute le informazioni essenziali per operare avrà luogo la prima modalita;

Verrà aperto in scrittura il file "raw_data.tsv" in cui saranno scritte le informazioni raccolte.

Verrà letto il file specificato da database_path fino alla fine andando ad incrementare l'offset ad ogni lettura di un numero di byte pari alla grandezza di una pagina (i database file sqlite sono suddivisi in pagine, solo la prima pagina conterrà l'header del database), così facendo ogni volta andremo a lavorare su una diversa pagina alla volta.

Ulteriori informazioni riguardo le pagine:
https://www.sqlite.org/fileformat.html#pages
https://www.sqlite.org/fileformat.html#b_tree_pages

Per ogni pagina verranno acquisite innanzitutto le informazioni dall'header.

Verrà letto il primo byte per capire il tipo di pagina e quindi la tipologia di dati contenuti all'interno. Se il byte convertito in int vale 13 allora la pagina in questione è di tipo (leaf table b-tree) ovvero dove sono contenuti i dati (records).

Successivamente verranno letti i seguenti due byte per capire a quanto ammonta l'offset al primo freeblock (i freeblock sono strutture utilizzate da SQLite per identificare lo spazio non allocato in una pagina).

I due byte seguenti rappresentano il numero di celle nella pagina, questa informazione ci occorrerà per calcolare il numero di byte occupati dal cell pointer array che è situato dopo l'header.

Nei due byte successivi troviamo l'offset all'inizio dell'area della pagina contenente le celle e al byte seguente il numero di byte liberi frammentati contenuti all'interno dell'area in cui sono le celle.

Ottenute queste informazioni possiamo calcolare la grandezza dell'area non allocata e l'offset dall'inizio della pagina all'inizio dell'area non allocata

L'immagine seguente può rendere meglio l'idea:

Header	Cell pointer array	Unallocated area	Cell area
8 byte	2 byte * numero di celle nella pagina	numero di byte variabile	numero di byte variabile

Inizio dell'area non allocata:
offset inizio pagina + 8 byte header + (2 byte * numero di celle nella pagina)

Lunghezza dell'area non allocata:
offset all'inizio dell'area della pagina contenente le celle - 8 byte header - (2 byte * numero di celle nella pagina)

Verrà letto il contenuto dell'area non allocata è memorizzato all'interno del file "raw_data.tsv". Prima di essere scritto sul file il contenuto verrà ripulito dei byte non printabili.

Come già accennato prima sqlite utilizza i freeblock per identificare lo spazio non allocato sparso all'interno della pagina, i freeblock sono organizzati come una catena di blocchi.

Dall'header della pagina abbiamo letto l'offset al primo freeblock, se il valore è diverso da 0 allora ci sposteremo all'offset indicato + l'offset dall'inizio del file alla pagina corrente.

Nei primi due byte è indicato l'offset al prossimo freeblock o 0 se non presente, nei successivi due byte la grandezza del freeblock attuale (inclusi i 4 byte letti).

Leggeremo il contenuto del freeblock e lo immagazzineremo nel file rimuovendone i caratteri non printabili.

Eseguiremo questi passaggi fino a quando l'offset al prossimo freeblock non sia pari a 0.

Terminata la modalità uno i dati delle aree non allocate verranno passate come input alla seconda modalità.

Prima di aver luogo la seconda modalità ha bisogno di sapere la struttura delle tabelle del database.

Tramite la funzione get_patterns presente all'interno del file patterns_extractor verrà aperto il database tramite il modulo sqlite3.

La funzione eseguirà una query sulla tabella sqlite_master per ottenere il nome di tutte le tabelle presenti all'interno del database.

Per ciascuna tabella ottenuta tramite l'istruzione PRAGMA table_info(nome_tabella) verranno ottenute le informazioni relative ai campi (nome, tipo_di_dato, primary_key).

Le informazioni ottenute verranno immagazzinate all'interno di un dizionario con questa struttura:

```
# esempio pattern[2] : tutte le tabelle che hanno due campi e la relativa struttura

# 2: {
#       'message_ftsv2_segments' :
#           [('blockid', 'INTEGER', 0), ('block', 'BLOB', 0)],
#      'message_ftsv2_docsize':
#           [('docid', 'INTEGER', 0), ('size', 'BLOB', 0)]
#    }
```

Il dizionario ci consentirà di trovare un insieme di tabelle candidate a seconda della struttura del record trovato.

Il informazioni sullo schema delle varie tabelle varranno inoltre utilizzate dalla classe report.

Seconda modalità:

Ogni area non allocata verrà passata alla funzione analyze_unallocated_area presente all'interno del file record_parser.py, la funzione per ogni valore != 0 farà partire la funzione "parse_varint_to_record" (perché potrebbe essere un possibile record o frammento di record).

Per ulteriori informazioni riguardo la struttura dei record:
https://www.sqlite.org/fileformat.html#record_format

La funzione parse_varint_to_record, per aver luogo utilizzerà inoltre l'area in bytes da analizzare convertita in variabili varint (un particolare tipo di dato utilizzato da sqlite per rappresentare numeri interi)

Ogni record in SQLite è composto da una variabile varint rappresentante il numero di bytes del payload, una seconda variabile varint rappresentante il row id e da un insieme di byte rappresentante il payload.

Il payload si suddivide in un header e un body, l'header inizia con una variabile varint rappresentante il numero di bytes nell'header, seguita da una o più varint (una per colonna) rappresentanti i serial type che ci indicano il tipo di dato in quella colonna e la size in byte del dato.

Terminato l’header inizia il body, tramite i serial type ottenuti precedentemente sapremo riconoscere le colonne nel body e quanti byte sono utilizzati per contenere il valore della relativa colonna.

Ottenuto un possibile record proveremo a vedere se ci sono tabelle candidate, ovvero seguenti la medesima struttura. Se la struttura è associata ad almeno ad una tabella, aggiungo il record ottenuto al dizionario della classe report, che ci consentirà successivamente di visualizzarlo tramite una comoda interfaccia.

Potenziali migliorie:

- I records rimossi potrebbero essere anche nelle freepage, quindi si potrebbe pensare di reperire le informazioni dalle freepage di tipo leaf table per provare a trovare ulteriori records.

- Potenziali records potrebbero essere presenti negli Hot Journals (file contenenti informazioni in grado di riportare il database in uno stato consistente in caso di crash).