InnoDB Datenbank Forensik
Rekonstruktion von Abfragen über Datenbank-interne Logfiles

Peter Frühwirt[+], Peter Kieseberg[+], Christoph Hochreiner[+], Sebastian Schrittwieser[*], Edgar Weippl[+]

[+] SBA Research gGmbH
(pfruehwirt,pkieseberg,chochreiner,eweippl)@sba-research.org

[*] Fachhochschule St. Pölten GmbH
sebastian.schrittwieser@fhstp.ac.at

Abstract: Datenbanksysteme werden in der digitalen Forensik oft vernachlässigt, hinterlassen aber durch Transaktionen an vielen Stellen (temporäre) Spuren. Jede Transaktion kann - sofern unterstützt - rückgängig gemacht werden (rollback). InnoDB, eine populäre MySQL-Storage-Engine, erzeugt hierfür eigene Log-Dateien. In dieser Arbeit dekodieren wir diese internen Log-Files von InnoDB, um die ausgeführten Datenbankabfragen zu rekonstruieren, welche Daten verändert haben. Abschließend gehen wir noch auf die Zusammenhänge zwischen der internen Datenspeicherung und den Log-Dateien ein.

1 Einführung

Immer wenn Daten verarbeitet werden, gibt es viele Bereiche, wo Informationen temporär gespeichert werden [PS08]. Forensische Analysen können vergangene Aktivitäten zeigen, Zeitabläufe bzw. Teilzeitabläufe erstellen und gelöschte Daten wiederherstellen [SML10]. Während diese Tatsache in der Computer-Forensik bereits sehr lange bekannt ist und viele Ansätze [YYA09, FTB+06, MKY08, JL03] und Software Lösungen [FC05] existieren, befindet sich die systematische Analyse von Datenbanksystemen erst in den Kinderschuhen [WB08, Oli09]. InnoDB ist dabei eine der wenigen transaktionssicheren MySQL-Storage-Engines mit Unterstützung von commits, rollbacks und crash recovery [Corc, WA02].

Jede Änderung von Daten ist mit gleichzeitiger Ausführung einer Mini-Transaktion (mtr) implementiert [Tuu09b] und führt daher zu zumindest einem Aufruf der Funktion `mtr_commit()`, welche Daten in die InnoDB redo logs schreibt. InnoDB speichert keine Daten in einem bestimmten Datenformat in die Logs, sondern schreibt nur Abbilder bestimmter Speicherbereiche [Cora], wodurch sich die Komplexität der Rekonstruktion erhöht.

Erschwerend kommt hinzu, dass InnoDB die Logs in den sogenannten DoubleWrite Buffer [KYL+12] schreibt, welcher zu einem späteren Zeitpunkt die Daten gesammelt in das Log-File einträgt. Desweiteren benutzt InnoDB seit MySQL 5.1 einen optimierten Ansatz für die Kompression [mys08] bei der Speicherung.

2 Aufbau der Logs

Beim erste Start von MySQL erzeugt InnoDB zwei Logdateien *ib_logfile0* und *ib_logfile1* mit einer standardmäßigen Größe von je fünf Megabyte, sofern InnoDB mit der Option `innodb_file_per_table` konfiguriert wurde [Cord]. Beide Dateien besitzen dieselbe Struktur und werden von InnoDB abwechselnd beschrieben. Ähnlich der internen Datenspeicherung [FHMW10] sind die Log-Files in mehrere Fragemente unterteilt, welche in den folgenden Abschnitten beschrieben werden.

2.1 Datei-Header

Der erste Teil des Log-Files ist der Datei-Header, welcher allgemeine Informationen zur Datei beinhaltet. Er besitzt eine fixe Länge von 0x14 Bytes.

Tabelle 1 zeigt ein Beispiel für einen typischen Datei-Header.

Offset	Länge	Wert	Bedeutung
0x00	4	00 00 00 00	Gruppennummer der Log-Datei
0x04	8	00 00 00 00 01 F1 EA 00	Erste Log Sequence Number (LSN) der Log-Datei
0x0C	4	00 00 00 00	Nummer des archivierten Log-Files (0 wenn Log-Datei nicht archiviert wurde)
0x10	32	20 20 20 20 00 00 00 00 00 00 00 [...] 00 00 00 00 00 00 00 00	Diese Bytes werden von InnoDB-Hot-Backup benutzt. Sie beinhalten den Wert "ibbackup" und den Zeitpunkt des erstellten Backups und werden für Informationen benutzt, welche dem Administrator angezeigt werden, wenn MySQL zum ersten Mal nach der Wiederherstellung gestartet wird.

Tabelle 1: Interpretation des Header-Blocks

2.2 Checkpoints und Crash recovery

InnoDB benutzt ein Checkpoint-System in den Log-Files und schreibt Modifikationen der Datenbank-Speicherseiten (*pages*) vom DoubleWrite Buffer in kleineren Schüben [KP05, Zai, BCKR02] auf die Festplatte, da eine Verarbeitung der Gesamtmenge die Ausführung

von SQL-Statements behindern würde.

InnoDB besitzt zwei Checkpoints in den Log-Files, welche abwechselnd beschrieben werden. Durch diese Methode wird garantiert, dass es immer mindestens einen vollständigen Checkpoint zur Wiederherstellung gibt. Während der Wiederherstellung nach einem Fehler [Tuu09a, Corb] lädt InnoDB die beiden Checkpoints und vergleicht deren ID-Labels. Jeder Checkpoint beinhaltet eine acht Byte lange *Log Sequence Number* (LSN). Diese LSN garantiert, dass alle Änderungen bis zu dieser Transaktionsnummer vorhanden sind. Dies bedeutet, dass alle Einträge kleiner als diese LSN auch im Log-File enthalten sind und kein Eintrag mit einer höheren ID existiert. Aus diesem Grund wird jede Änderung, welche nicht auf die Festplatte gespeichert wurde, in die crash-recovery-logs beziehungsweise in die Rollback-Logs gespeichert, InnoDB muss vor dem Schreiben auf die Festplatte einen Checkpoint erstellen [Tuu09a].
Die beiden Checkpoints befinden sich in den Log-Files *ib_logfile0* und *ib_logfile1* unter der Adresse 0x200 beziehungsweise 0x400. Jeder Checkpoint besitzt dieselbe Struktur und eine fixe Länge von 0x130 Bytes. Tabelle 3 zeigt eine detailierte Darstellung des Aufbaus eines Checkpoint-Blocks.

Die Daten des Checkpoints werden von der Methode `log_group_checkpoint()` [Tuu09b] (line 1675-1795) in den log group header geschrieben.

Offset	Länge	Wert	Bedeutung
0x00	8	00 00 00 00 00 00 00 12	Log checkpoint number
0x08	8	00 00 00 00 01 F3 64 92	Log sequence number des Checkpoints
0x10	4	00 00 0A 92	Offset zum log entry, berechnet durch `log_group_calc_lsn_offset()` [Tuu09b]
0x14	4	00 10 00 00	Größe des Buffers (fixer Wert: $2 \cdot 1024 \cdot 1024$)
0x18	8	FF FF FF FF FF FF FF FF	Archivierte log sequence number. Sofern `UNIV_LOG_ARCHIVE` nicht aktiviert ist, setzt InnoDB diese Felder auf FF FF FF FF FF FF FF FF.
0x20	288	00 00 [...] 00	Spacing und padding
0x120	4	28 E2 C3 AC	Prüfsumme 1 des gesamten Checkpoint-Blocks: 0x00-0x19F)
0x124	4	01 B0 72 29	Prüfsumme 2 über den Checkpoint-Block ohne LSN inklusive Prüfsumme 1: 0x08-0x124
0x128	4	00 00 00 05	Derzeitiges fsp (filespace) free limit im tablespace 0 (Einheit: MB). Diese Bytes werden nur von ibbackup benutzt, um zu entscheiden ob unbenutze Enden von non-auto-extending Datenfeldern abgeschnitten werden können [Tuu09c].
0x12C	4	55 E7 71 8B	Dieser Fixwert definiert, dass dieser Checkpoint die obigen Felder beinhaltet und wurde mit InnoDB-3.23.50 eingeführt [Tuu09c].

Tabelle 3: Interpretation des Checkpoints

2.3 Log Blöcke

Die Einträge in den Logdateien sind - im Gegensatz zu den Daten - nicht in pages sondern in Blöcken organisiert. Jeder Block besteht aus 512 Bytes Daten. Die Größe ist mit der Größe des Disk-Sektors identisch [Zai09]. Jeder Block besteht aus drei Teilen: Dem Log-Header, den eigentlichen Daten und dem Trailer. Diese Struktur dient der Erhöhung der Performance bei der Navigation in den Log-Files.

2.3.1 Log Block Header

Die ersten 12 Bytes eines Blocks werden *log block header* genannt. Dieser Block beinhaltet alle Informationen, die von InnoDB zur Verwaltung und zum Lesen der Log-Daten benötigt werden. Alle 1000 Bytes erstellt InnoDB automatisch einen neuen Header, daher gibt es in jedem Header einen Offset zum ersten Log-Eintrag innerhalb des nachfolgenden Datenblocks. InnoDB unterbricht bestehende Log-Einträge, um den Header an die dafür vorgesehene Stelle zu schreiben. Dies ermöglicht InnoDB schneller zwischen den Blöcken zu navigieren.

Offset	Länge	Wert	Bedeutung
0x00	4	80 00 F9 B2	Nummer des Log block headers. Wenn das Most Significant Bit (MSB) auf 1 gesetzt ist, ist der nachfolgende Block der erste Block im Log File. [Tuu09c]
0x04	2	00 EE	Anzahl an geschriebenen Bytes in diesem Block.
0x06	2	00 0C	Offset zum ersten Log-Eintrag in diesem Block.

Tabelle 4: Interpretation des log block headers

2.3.2 Log Block Trailer

Der Log Block Trailer (LBT) besteht aus einer vier Byte langen Prüfsumme zur Verifikation der Gültigkeit des Blocks.

Offset	Länge	Wert	Bedeutung
0x00	4	36 9E 2E 96	Prüfsumme des Log Block Inhaltes. In älteren InnoDB Versionen (vor Version 3.23.52) wurde an dieser Stelle derselbe Wert wie `LOG_BLOCK_HDR_NO` anstelle der Prüfsumme gespeichert [Tuu09c].

Tabelle 5: Interpretation des log block trailers

3 Log-Einträge

Die nachfolgenden Demo-Rekonstruktionen der Datenbankabfragen basieren auf dem folgenden Datenmodell (Listing 1).

Listing 1: Verwendete Tabellenstruktur

```
CREATE TABLE `fruit3` (
  `primaryKey` int(10) NOT NULL,
  `field1` varchar(255) NOT NULL,
  `field2` varchar(255) NOT NULL,
  `field3` varchar(255) NOT NULL,
  PRIMARY KEY (`primaryKey`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Wir haben zwei einfache Datentypen (Integer und Varchar) benutzt, um die Vorgehensweise bei der Rekonstruktion zu demonstrieren. InnoDB speichert einen Integer-Wert in einem Feld mit einer fixen Länge von 4 Bytes, das Datenformat Varchar ist hingegen von variabler Länge. Diese beiden Datentypen decken damit die Gestalt der Logfiles für nahezu alle anderen Typen ab und unterscheiden sich von diesen lediglich in der Länge des verwendeten Speicherplatzes und der Interpretation der Daten. Für unsere forensische Analyse ist das Wissen über die Tabellenstruktur erforderlich. Dieses Wissen kann aus den Informationen der `.frm` Dateien [FHMW10] bzw. den Speicherdumps aus den Log-Files gewonnen werden.

4 Rekonstruktion von SQL-Queries

In diesem Abschnitt werden SQL-Statements mit Hilfe der InnoDB-Log-Files rekonstruiert. Zu beachten ist, dass einige Daten in komprimierter Form gespeichert sind. Die Längenangaben dieser Werte sind in den Tabellen mit Sternen versehen und können daher abhängig vom konkreten SQL-Statement teilweise erheblich abweichen. Ältere Versionen von InnoDB schrieben die Logs ohne Komprimierung und verbrauchten dadurch wesentlich mehr Speicher. Auf die genauere Definition und Analyse dieses Dateiformats für forensische Zwecke wird in dieser Arbeit nicht eingegangen.

Für jede SQL-Abfrage, welche Daten ändert, erzeugt InnoDB zumindest einen `MLOG_UNDO_INSERT` Log-Eintrag. Dieser Eintrag speichert die betroffene Table Identification Number (TIN), den Typ der Abfrage (INSERT, UPDATE, DELETE) sowie typspezifische Informationen.

4.1 Insert

InnoDB erstellt für eine einzelne INSERT-Abfrage insgesamt 9 Log-Einträge. Diese sind alle mit MLOG_ als Prefix versehen.

1. 8BYTES
2. UNDO_HDR_REUSE
3. 2BYTES
4. 2BYTES
5. 2BYTES
6. MULTI_REC_END
7. UNDO_INSERT
8. COMP_REC_INSERT
9. 2BYTES

Die meisten dieser Log-Einträge sind dabei vom Typ MLOG_2BYTES beziehungsweise MLOG_8BYTES und zeigen lediglich kleinere Änderungen in Dateien an. Für die forensische Analyse sind nur die Einträge MLOG_UNDO_INSERT und MLOG_COMP_REC_INSERT interessant.

In diesem Abschnitt beschreiben wir die letzten Bestandteile des Log-Eintrags MLOG_COMP_REC_INSERT. Im Vorhinein wurde bereits die referenzierte Tabelle im MLOG_UNDO_INSERT-Eintrag ausgelesen. Die Tabellenstruktur ist daher bereits vor dem Auslesen der eigentlichen Logdaten, über die Referenz zur manipulierten Tabelle, bekannt.

Offset	Länge	Wert	Bedeutung
0x00	1	26	Log-Typ des Eintrags (COMP_REC_INSERT)
0x01	1*	58	Tablespace-ID
0x02	1*	03	Page-ID
0x03	2	00 06	Anzahl der Felder in diesem Eintrag s (n)
0x05	2	00 01	Anzahl der einzigartigen Felder (n_unique)
0x07	2	80 04	Länge des ersten Datenfeldes (primaryKey).
0x09	2	80 06	Länge des zweiten Datenfeldes (transactionID)
0x0B	2	80 07	Länge des dritten Datenfeldes (data rollback pointer)
0x0D	2	80 00	Länge des vierten Datenfeldes (Zu beachten ist, dass bei diesem Datenfeld keine Längenangabe existiert, da es sich um ein Feld von dynamischer Länge handelt (Varchar). Das hat zur Folge, dass es im späteren Verlauf noch eine Größenangabe für dieses Feld geben muss)
0x0F	2	80 00	Länge des fünften Datenfelds (field2)
0x11	2	80 00	Länge des sechsten Datenfelds (field3)
0x13	2	00 63	Offset
0x15	1*	59	Länge des letzten Segments (*end segments*)
0x16	1	00	Info- und Status-Bits
0x17	1*	08	Offset zum Ursprung
0x18	1	00	Mismatch index
0x19	1*	04	Länge des ersten Felds mit variabler Länge (field1)
0x1A	1*	05	Länge des zweiten Felds mit variabler Länge (field2)
0x1B	1*	0A	Länge des dritten Felds mit variabler Länge (field3)
0x1C	5	00 00 30 FF 56	Unbekannt
0x21	4	80 00 00 04	PrimaryKey
0x25	6	00 00 00 00 40 01	Transaction-ID
0x2B	7	00 00 00 00 33 21 28	Data rollback pointer

0x31	10*	73 74 72 61 77 62 65 72 72 79	Daten des ersten Felds (”strawberry”)
0x3B	5*	61 70 70 62 65	Daten des zweiten Felds (”apple”)
0x40	4*	6B 69 77 69	Daten des dritten Felds (”kiwi”)

Die Struktur des `MLOG_COMP_REC_INSERT` Eintrags ist sehr kompliziert. Nach dem allgemeinen Datenblock (logType, spaceID und pageID), welcher die verwendete Tabelle beinhaltet, finden sich die folgenden Längeninformationen: n und n_unique. Der Wert n gibt an, aus wie vielen Feldern dieser Log-Eintrag besteht, n_unique definiert die Anzahl der Primärschlüssel. Anschließend an diese beiden Feldern folgen jeweils n-mal 2 Bytes Längeninformationen zu den jeweiligen Datenfeldern, wobei die ersten für die Primärschlüssel reserviert sind (n_unique Felder).

Sollte die Feldlänge den Wert 0x8000 besitzen, bedeutet dies, dass die Feldlänge dynamisch berechnet wird und vom Datensatz abhängig ist (zum Beispiel im Fall eines Feldes vom Typ VARCHAR). Für alle statischen Datentypen (z.B. INTEGER) definiert dieser Wert die Länge des Feldes. Zusätzlich muss beachtet werden, dass in diesen Längendefinitionen auch systeminterne Felder, wie die Transaktions-ID und der data rollback pointer definiert werden.

In den nächsten sieben* Feldern befinden sich Metadaten, die für die Rekonstruktion nicht relevant sind. Darauf anschließend findet sich wieder Informationen, welche für die Berechnung der Länge des Datensatzes entscheidend sind. Die tatsächliche Länge wird hier für jeden Eintrag mit dynamischer Länge spezifiziert.

Die folgenden fünf Bytes werden für diverse flags benötigt. In den ersten drei Blöcken befinden sich die eigentlichen Log-Daten, beginnend mit dem Primärschlüssel. Die Anzahl der Felder des Primärschlüssels s wird dabei am Anfang im Feld n_unique definiert. Durch die Typinformationen aus der .frm-Datei [FHMW10] ist bekannt, dass es sich hierbei um einen Primärschlüssel bestehend aus einer INTEGER-Spalte handelt. Anhand dieser Information kann die Länge und der enthaltene Wert berechnet werden.

Die nächsten beiden zwei Felder sind systeminterne Felder: Die Transaktions-ID und der Data Rollback Pointer (DRP). Der DRP referenziert die Position in der Datei, an welche der Datensatz geschrieben wurde. Wenn zum Beispiel ein Dateneintrag mit Hilfe einer UPDATE-Operation verändert wird, besitzt dieser Log-Eintrag denselben data rollback pointer wie der Ursprungseintrag. Die letzten Felder beinhalten die eigentlichen Daten der jeweiligen Spalten, welche nicht Teil des Primärschlüssels sind. Die Anzahl der Felder ist immer (n_unique - 2), da die beiden internen Felder (DRP und Transaktions-ID) bereits behandelt wurden. In dem angeführten Beispiel besitzen die Felder eine dynamisch allokierte Länge (Typ: VARCHAR). Aus diesem Grund wird die Länge nicht ausschließlich in den Feldern über die Längenangaben definiert (Einträge nach dem Feld n_unique). Es gibt das zuvor erwähnte Feld für die dynamischen Längen. Mit Hilfe dieser Informationen können die Daten der Felder rekonstruiert werden: ”strawberry”, ”apple”, ”kiwi”.

Wir kennen daher den Primärschlüssel sowie die Werte der Datenfelder, in Kombination mit den gesammelten Informationen über die Tabellenstruktur aus der .frm-Datei kann die ausgeführte Datenbankabfrage wiederherstellt werden (Listing 2).

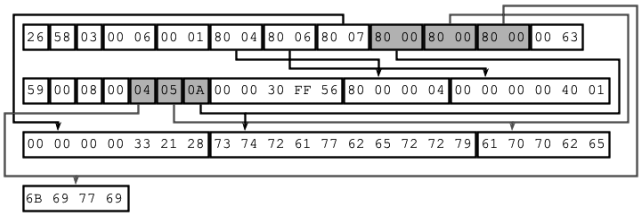

Abbildung 1: Zusammenhänge zwischen den Datenfeldern in einem MLOG_COMP_REC_INSERT Log-Eintrag

Listing 2: Wiederhergestellte INSERT-Abfrage

```
INSERT INTO forensic1.fruit3 (primaryKey, field1, field2, field3)
       VALUES (4, 'strawberry','apple','kiwi');
```

4.2 Update

Die Rekonstruktion einer UPDATE-Operation verläuft ähnlich wie die eines INSERT-Statements. Der einzige Unterschied ist, dass sich zusätzliche, für die forensische Analyse interessante Informationen extrahieren lassen. In unseren Untersuchungen haben wir nur UPDATE-Operationen betrachtet, die den Primärschlüssel nicht verändern, da diese keine Änderungen am Index vornehmen und dadurch weitere Änderungen in den Log-Dateien verursachen.

4.2.1 Wiederherstellung von überschriebenen Daten

InnoDB speichert die überschriebenen Daten für die Wiederherstellung nach einem Ausfall der Datenbank oder zur Durchführung von Rollbacks. Zunächst muss für die Analyse dieser Daten der `MLOG_UNDO_INSERT` Eintrag betrachtet werden.

Offset	Länge	Wert	Bedeutung
0x00	1	94	Typ des Eintrags (UNDO_INSERT)
0x01	1*	00	Tablespace-ID
0x02	1*	33	Page-ID
0x03	2	00 1B	Länge des Log-Eintrags
0x05	1	1C	Typ der Datenmanipulation (Update eines bestehenden Eintrags)
0x06	2	00 68	ID der Tabelle
0x08	6	00 00 00 00 40 01	Letzte Transaktionsnummer des veränderten Feldes
0x0D	5*	00 00 33 21 28 04	Bisheriger DRP
0x13	1	04	Länge des Primärschlüssels
0x14	4	80 00 00 04	Betroffener Primärschlüssel

0x18	1	01	Anzahl der geänderten Felder
0x19	1	04	FieldID
0x1A	1	05	Länge des Feldes
0x1B	5	61 70 70 6C 65	Überschriebener Wert des Feldes

Die ersten drei Bytes sind wie in jedem Log-Eintrag gleich aufgebaut: Typ, SpaceID und PageID, die Bytes vier und fünf definieren die Länge des Log-Eintrags, daher ist das Ende des Eintrags sehr einfach zu bestimmen. Die folgenden Bytes definieren den Typ des *undo-insert*-Typs. Diese Klassifikation ist notwendig, da der *undo-insert*-Typ nur besagt, dass irgendetwas in einem Eintrag geändert wurde. Die folgenden Daten sind abhängig von diesem Typ und müssen gesondert interpretiert werden. In dem vorhandenen Beispiel betrachten wir nun den Typ 12 (`TRX_UNDO_UPDATE_EXIST_RECORD`) näher. Der Typ wird berechnet, indem der Wert modulo 16 genommen wird.

Die ersten beiden Bytes bestimmen die betroffene Tabelle (table identifier). Dieses Identifizierungsmerkmal findet man auch in den Tabellendefinitionen der `.frm`-Dateien an der Adresse 0x26. In Kombination mit diesen Informationen erhält man den Namen der Tabelle. In dem vorhandenen Beispiel handelt es sich um die Tabelle ”fruit3”. Die folgenden sechs beziehungsweise sieben Bytes beinhalten die Transaktions-ID und den DRP der Datenfelder. Die Transaktions-ID referenziert auf die letzte ausgeführte Transaktion vor diesem Update. In diesem Beispiel handelt es sich um die INSERT-Operation, die diesen Eintrag angelegt hat. Mit Hilfe dieser Transaktionsnummer kann die komplette Historie eines Datensatzes, beginnend bei seiner Erstellung, rekonstruiert werden.

In den nachfolgenden Bytes wird der Primärschlüssel definiert. Abhängig von der Anzahl der Schlüssel wird dieser Schritt mehrfach wiederholt. Daher ist es wichtig vorab zu wissen, aus wie vielen Feldern der Primärschlüssel der Tabelle besteht. Das erste Byte definiert die Länge des Primärschlüsselfeldes. In diesem Beispiel handelt es sich um einen Integer, daher beinhaltet dieses Feld den Wert vier. Die nächsten *n* Bytes, wobei n von der vorherigen Längenangabe bestimmt wird, speichern den konkreten Wert des Primärschlüsselfeldes: vier ($0x8004$ *signed*).
Die nächsten Bytes bestimmen die Anzahl der Spalten m (in dem vorhandenen Beispiel ist $m = 1$), die durch die UPDATE-Operation verändert wurden.

Aus diesem Grund muss der nachfolgende Schritt *m*-mal wiederholt werden. Dabei besitzt jeder Datenblock dieselbe Struktur: Das erste Byte speichert die Feldnummer, das zweite Feld die Länge des Datenblocks und die nachfolgenden Bytes die eigentlichen Daten. In unserem Beispiel wurde der Wert von Feld Nr. 4 (”field2”) mit dem vorherigen Wert 0x61 70 70 6C 65 (”apple”) überschrieben.

4.3 Delete

Die Wiederherstellung von DELETE-Operationen verläuft ähnlich wie bei UPDATE Operationen. Es muss zwischen DELETE-Operationen, welche einen Datensatz wirklich physikalisch löschen und solchen, die den Datensatz nur als gelöscht markieren unterschieden werden.

Ein Log-Eintrag, der einen Eintrag lediglich als gelöscht markiert, ist sehr kurz und besteht aus 4 Teilen. Für die forensische Rekonstruktion ist hier jedoch nur ein einziger Block interessant: `MLOG_UNDO_INSERT`. Die folgende Tabelle zeigt diesen Log-Eintrag für eine durchgeführte DELETE-Operation.

Offset	Länge	Wert	Bedeutung
0x00	1	94	Typ des Log-Eintrags (UNDO_INSERT)
0x01	1*	00	Tablespace-ID
0x02	1*	33	Page-ID
0x03	2	00 1E	Länge des Log-Eintrags
0x05	1	0E	Typ der Datenmanipulation (delete record)
0x06	2	00 66	Table-ID
0x08	6	00 00 00 00 28 01	Letzte Transaktionsnummer des gelöschten Feldes
0x0D	8*	E0 80 00 00 00 2D 01 01 10	Bisheriger DRP
0x17	1	04	Länge des Primärschlüssels
0x18	4	80 00 00 04	Betroffener Primärschlüssel
0x1B	3	00 08 00	Unknown
0x1E	1	04	Länge des Primärschlüsselfeldes
0x20	4	80 00 00 01	Primärschlüssel des gelöschten Eintrages

Analog zu den UPDATE-Operationen muss die Anzahl der Primärschlüssel bereits bekannt sein. Dieser Log-Eintrag speichert im Prinzip nur den Primärschlüssel des gelöschten Datensatzes und kann über das Dateisystem wiederhergestellt werden [FHMW10]. Der komplette Datensatz wird nur dann im Log gespeichert, wenn der Datensatz physisch gelöscht wird. Die ausgeführte Abfrage kann analog zu den INSERT- bzw. UPDATE-Operationen wiederhergestellt werden.

Listing 3: Rekonstruierte DELETE-Query

```
DELETE FROM forensic1.fruit3 WHERE primaryKey=1;
```

5 Fazit

InnoDB verwendet interne Log-Dateien, um Transaktionen zurückzusetzen (Rollback) beziehungsweise zur Wiederherstellung nach einem Crash. Die gespeicherten Log-Einträge beinhalten Kopien von Teilen des Speicherbereichs sowie von Daten. In dieser Arbeit wurde gezeigt, wie SQL-Operationen aus den internen Log-Dateien der Datenbank wiederhergestellt werden können. Die demonstrierten Techniken können genutzt werden, um eine komplette Transaktionsgeschichte zu rekonstruieren. Wir haben eine bekannte Struktur der Tabellen vorausgesetzt, welche wir durch unsere bisherigen Arbeiten [FHMW10, FKS+13] bereits forensisch gewinnen konnten. Zusätzlich wurden Operationen, die die Tabellenstruktur verändern (z.B. ALTER TABLE), vernachlässigt. Durch die gewonnenen Informationen aus den Log-Dateien lassen sich diese Informationen jedoch analog wiederherstellen.

Literatur

[BCKR02] Ryan Bannon, Alvin Chin, Faryaaz Kassam und Andrew Roszko. InnoDB Concrete Architecture. *University of Waterloo*, 2002.

[Cora] Oracle Corporation. Compression and the InnoDB Log Files (accessed 31.01.2014). `http://dev.mysql.com/doc/innodb-plugin/1.0/en/innodb-compression-internals-log.html`.

[Corb] Oracle Corporation. InnoDB Checkpoints (accessed 31.01.2014). `http://dev.mysql.com/doc/mysql-backup-excerpt/5.1/en/innodb-checkpoints.html`.

[Corc] Oracle Corporation. Storage Engines (accessed 31.01.2014). `http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html`.

[Cord] Oracle Corporation. Using Per-Table Tablespaces (accessed 31.01.2014). `http://dev.mysql.com/doc/refman/5.5/en/innodb-multiple-tablespaces.html`.

[FC05] Guillermo Francia und Keion Clinton. Computer forensics laboratory and tools. *Journal of Computing Sciences in Colleges*, 20(6), June 2005.

[FHMW10] Peter Frühwirt, Markus Huber, Martin Mulazzani und Edgar Weippl. InnoDB Database Forensics. In *Proceedings of the 24th International Conference on Advanced Information Networking and Applications (AINA 2010)*, 2010.

[FKS+12] Peter Frühwirt, Peter Kieseberg, Sebastian Schrittwieser, Markus Huber und Edgar Weippl. InnoDB Database Forensics: Reconstructing Data Manipulation Queries from Redo Logs. In *5th International Workshop on Digital Forensic*, 2012.

[FKS+13] Peter Frühwirt, Peter Kieseberg, Sebastian Schrittwieser, Markus Huber und Edgar Weippl. InnoDB database forensics: Enhanced reconstruction of data manipulation queries from redo logs. *Information Security Technical Report*, 2013.

[FTB+06] Guillermo Francia, Monica Trifas, Dorothy Brown, Rahjima Francia und Chrissy Scott. Visualization and management of digital forensics data. In *Proceedings of the 3rd annual conference on Information security curriculum development*, 2006.

[JL03] Hongxia Jin und J. Lotspiech. Forensic analysis for tamper resistant software. *14th International Symposium on Software Reliability Engineering*, November 2003.

[KP05] Michael Kruckenberg und Jay Pipes. *Pro MySQL*. Apress, 2005.

[KYL+12] Woon-Hak Kang, Gi-Tae Yun, Sang-Phil Lim, Dong-In Shin, Yang-Hun Park, Sang-Won Lee und Bongki Moon. InnoDB DoubleWrite Buffer as Read Cache using SSDs. In *10th USENIX Conference on File and Storage Technologies*, 2012.

[MKY08] J. Todd McDonald, Yong Kim und Alec Yasinsac. Software issues in digital forensics. *ACM SIGOPS Operating Systems Review*, 42(3), April 2008.

[mys08] Changes in Release 5.1.x (Production). http://dev.mysql.com/doc/refman/5.1/en/news-5-1-x.html, 2008.

[Oli09] Martin Olivier. On metadata context in Database Forensics. *Digital Investigation*, 4(3-4), March 2009.

[PS08] Kyriacos Pavlou und Richard Snodgrass. Forensic analysis of database tampering. *ACM Transactions on Database Systems (TODS)*, 33(4), November 2008.

[SML10] Patrick Stahlberg, Gerome Miklau und Brian Neil Levin. Threats to privacy in the forensic analysis of database systems. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2010.

[Tuu09a] Heikki Tuuri. Crash Recovery and Media Recovery in InnoDB. In *MySQL Conference*, April 2009.

[Tuu09b] Heikki Tuuri. MySQL Source Code (5.1.32). /src/storage/innobase/log/log0log.c, 2009.

[Tuu09c] Heikki Tuuri. MySQL Source Code (5.1.32). /src/storage/innobase/include/log0log.h, 2009.

[WA02] Michael Widenius und David Axmark. *MySQL reference manual: documentation from the source*. O'Reilly, 2002.

[WB08] Paul Wright und Donald Burleson. *Oracle Forensics: Oracle Security Best Practices (Oracle In-Focus series)*. Paperback, 2008.

[YYA09] Pei-Hua Yen, Chung-Huang Yang und Tae-Nam Ahn. Design and implementation of a live-analysis digital forensic system. In *Proceedings of the 2009 International Conference on Hybrid Information Technology*, 2009.

[Zai] Peter Zaitsev. MySQL Performance Blog - InnoDB Double Write (accessed 31.01.2014). `http://www.mysqlperformanceblog.com/2006/08/04/innodb-double-write/`.

[Zai09] Peter Zaitsev. Innodb Architecture and Performance Optimization. In *O'Reilly MySQLConference and Expo*, April 2009.