KIPS Tr. Comp. and Comm. Sys.
Vol.6, No.2 pp.59~66 pISSN: 2287-5891

https://doi.org/10.3745/KTCCS.2017.6.2.59

The Recovery Method for MySQL InnoDB Using Feature of IBD Structure

Jeewon Jang† · Doowon Jeoung†† · Sang Jin Lee†††

ABSTRACT

MySQL database is the second place in the market share of the current database. Especially InnoDB storage engine has been used in the default storage engine from the version of MySQL5.5. And many companies are using the MySQL database with InnoDB storage engine. Study on the structural features and the log of the InnoDB storage engine in the field of digital forensics has been steadily underway, but for how to restore on a record-by-record basis for the deleted data, has not been studied. In the process of digital forensic investigation, database administrators damaged evidence for the purpose of destruction of evidence. For this reason, it is important in the process of forensic investigation to recover deleted record in database. In this paper, We proposed the method of recovering deleted data on a record-by-record in database by analyzing the structure of MySQL InnoDB storage engine. And we prove this method by tools. This method can be prevented by database anti forensic, and used to recover deleted data when incident which is related with MySQL InnoDB database is occurred.

Keywords : Database Forensic, The Database Recovery in Record, Digital Forensic, InnoDB Storage Engine, MySQL, MySQL Forensic

IBD 구조적 특징을 이용한 MySQL InnoDB의 레코드 복구 기법

장 지 원† · 정 두 원†† · 이 상 진†††

요 약

MySQL 데이터베이스는 현재 데이터베이스 시장 점유율에서 2위를 차지하고 있다. 특히 InnoDB 스토리지 엔진은 MySQL 5.5 버전부터 디폴트 스토리지 엔진으로 사용되어 왔으며, 많은 기업에서 InnoDB 스토리지 엔진으로 MySQL 데이터베이스를 사용하고 있다. 디지털 포렌식 분야에서 InnoDB 스토리지 엔진에 대한 구조적 특징과 로그에 관한 연구는 꾸준히 진행되어 왔으나, 삭제된 데이터에 대해 레코드 단위로 복구하는 방법에 대해서는 연구되지 않았다. 기업 조사 시 데이터베이스 관리자가 사전에 증거 인멸을 목적으로 데이터를 훼손하는 경우가 많으므로 이를 복구하는 것은 포렌식 수사 과정에서 중요하다. 본 논문에서는 MySQL InnoDB 스토리지 엔진의 구조를 분석하여 삭제된 데이터를 레코드 단위로 복구하는 기법을 제안하고 제작한 도구를 활용하여 이를 검증한다. 이는 디지털 포렌식 관점에서 데이터베이스 안티포렌식 행위에 대해 대비할 수 있으며, MySQL InnoDB 데이터베이스와 관련된 사건 발생시, 고의로 삭제된 데이터를 복구하는 데 활용할 수 있다.

키워드 : 데이터베이스 포렌식, 레코드 단위 복구, 디지털 포렌식, InnoDB 스토리지 엔진, MySQL, MySQL 포렌식

1. 서 론

데이터베이스는 방대한 데이터를 효율적으로 관리하기 위해 다양한 기관에서 사용되고 있다. 각 기관의 업무데이터, 개인정보와 같은 중요한 정보를 통합적으로 관리하기 위해 데이터베이스에 저장하고 있으며, 이는 데이터베이스와 관련된 사건이 발생했을 경우 디지털 포렌식 관점에서 결정적인 증거가 나올 수 있는 중요한 조사대상임을 알 수 있다[1].

현재 많은 수사과정에서 데이터베이스에 대한 수사가 활발하게 이루어지는데, 피조사자가 데이터베이스에서 결정적인 데이터를 삭제하여, 디지털 증거를 훼손하는 경우가 많이 발생하고 있다. 이에 데이터베이스 내에서 삭제된 데이터를 복구하는 것이 디지털 포렌식 수사과정에서 중요해지고 있다[2].

※ 이 논문은 2016년도 정부(미래창조과학부)의 재원으로 한국연구재단-공공복지안전사업의 지원을 받아 수행된 연구임(2012M3A2A1051106).
† 준 회 원 : 고려대학교 정보보호대학원 정보보호학과 석사과정
†† 비 회 원 : 고려대학교 정보보호대학원 정보보호학과 석·박사통합과정
††† 종신회원 : 고려대학교 정보보호대학원 교수
Manuscript Received : August 2, 2016
First Revision : November 8, 2016
Accepted : November 21, 2016
* Corresponding Author : Sang Jin Lee(sangjin@korea.ac.kr)

MySQL 데이터베이스는 현재 전 세계 점유율 2위를 차지할 만큼 다양한 기관에서 사용하고 있다. 특히 오픈소스로 배포되기 때문에 중, 소기업체에서 많이 사용되고 있다. MySQL의 주요 스토리지 엔진(Storage Engine)으로 MyISAM과 InnoDB 엔진이 있다. 이 중 InnoDB엔진은 MySQL 5.5 버전 이상에서 디폴트 스토리지 엔진으로 사용되고 있으며, 최근 배포된 MySQL 5.7.13 버전까지 사용되고 있다[7].

InnoDB 스토리지 엔진의 포렌식 관점에서의 구조적 특징과 Commit, Rollback에 관한 로그에 관련된 연구가 활발하게 진행되어 왔으나[4], 수사에 도움이 될 수 있는 실질적인 레코드단위 복구에 대한 연구는 진행되지 않았다. 본 논문에서는 MySQL 데이터베이스에서 InnoDB 스토리지 엔진의 구조를 분석하고, 구조적 특징을 이용하여 삭제된 레코드에 대한 복구기법을 소개한다. 본 연구는 MySQL 5.6-5.7.13 버전을 대상으로 진행되었으며, 제시한 복구 기법으로 제작된 도구로 실험한 결과를 통해 검증한다.

2. 관련 연구

현재까지 MySQL 데이터베이스의 InnoDB 스토리지 엔진에 대한 연구는 활발하게 이루어져 왔다. Fruhwirt[4]은 MySQL 5.1 버전을 기준으로 InnoDB 엔진의 Redo logs에서 사용자가 과거에 행했던 데이터 삭제 및 업데이트 등의 쿼리를 복구하여 확인하는 방법에 대해 연구하였다. 이는 사용자의 행위를 추적한다는 것에서 의미가 있지만 실제 Redo 영역에 남겨지는 로그에는 일시적으로 남는다는 것에서 한계가 존재한다.

Fruhwirt와 Huber(2010)의 연구는 MySQL 5.1.32 버전을 기준으로 InnoDB 엔진을 사용한 MySQL 데이터베이스의 구조를 포렌식 관점에서 분석하였다[3]. 해당 논문에서는 InnoDB 엔진에서 생성되는 FRM 파일에 대한 구조를 통해 스키마에 대한 복구 관점과, 각 데이터 타입의 저장 방식에 대해 연구가 진행되었다. 그러나 InnoDB 스토리지 엔진에서 실제 사용자의 데이터가 저장되는 곳은 IBD 파일이며, IBD 파일 구조에 대한 연구와 실제 사용자가 삭제한 레코드 복구에 관한 연구는 진행되지 않았다.

Noh(2016)의 논문에서는 MySQL MyISAM 데이터베이스에서 삭제된 레코드 복구에 대한 연구가 진행되었다[5]. 저자는 MySQL 데이터베이스의 디렉토리 구조와 MyISAM 스토리지 엔진에서 생성되는 데이터 파일(FRM, MYI, MYD) 구조를 분석하였고, 이를 통해 MySQL MyISAM에서 삭제된 레코드 복구 기법에 대해 연구하였다. 데이터베이스 디렉토리 구조와, 데이터베이스 내에 생성된 테이블의 스키마 정보가 저장된 FRM 파일 구조는 InnoDB 데이터베이스와 동일하다. 하지만 MySQL 5.5 버전 이후부터 InnoDB 스토리지 엔진이 디폴트로 사용되고 있으며 MyISAM에서 저장되는 방식이 InnoDB와 상이하기 때문에, InnoDB 데이터베이스에서 레코드 단위로 복구하는 것에 대한 추가적인 연구가 필요하다.

3. 배경 지식

3.1 MySQL InnoDB 디렉토리 구조

MySQL의 데이터 디렉토리는 서버로부터 관리되는 모든 데이터베이스를 포함한다. MySQL서버에서 관리되는 데이터베이스는 C:\ProgramData\MySQL\data 경로에 디렉토리로 존재한다(Fig. 1).

MySQL에서 테이블에 관련된 파일들은 데이터베이스 디렉토리 내에 존재한다. 데이터베이스 수집은 MySQL 데이터 디렉토리 또는 특정 데이터베이스의 디렉토리 전체를 복사하는 방법으로 획득 가능하고, 활성 상태의 파티션에서 MySQL 서비스가 동작 중인 경우에도 복사가 가능하다.

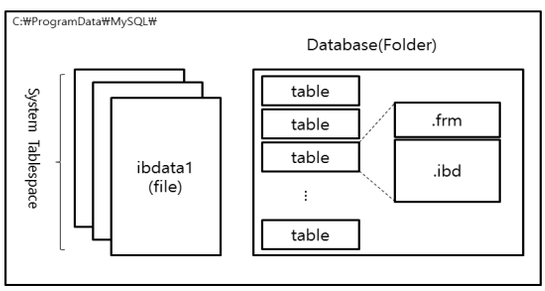

Fig. 1. The Structure of MySQL Directory

3.2 FRM 파일 구조

FRM 파일은 MySQL 데이터베이스에서 테이블의 스키마 구조 정보를 저장한다. MyISAM과 InnoDB 스토리지 엔진 모두 공통적으로 스키마 구조 정보에 대한 파일로 FRM 파일 포맷을 사용한다.

FRM 파일 포맷은 Fig. 2와 같이 크게 헤더(Header), 키와 인덱스 정보 영역(Key and Index information area), 컬럼 메타데이터 영역(Column metadata area)으로 나뉜다. 헤더 영역에서 테이블의 일반적인 정보를 확인할 수 있으며, 키와 인덱스 정보영역에서 기본키(Primary Key)와 인덱스(Index)가 설정된 키에 대한 정보를 확인할 수 있다.

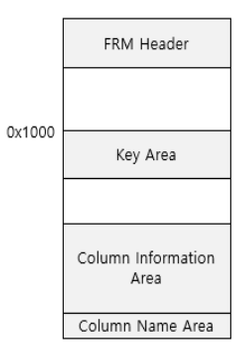

Fig. 2. The Layout .frm File

3.2.1 FRM 파일의 헤더

FRM 파일 헤더의 실제 사용되는 크기는 70바이트이며, IBD 파일에서의 삭제된 레코드 분석에 필요한 정보는 스토

리지 엔진 유형(Storage Engine Type), 레코드 포맷(Record Format), 테이블 문자세트(Character Set ID), 컬럼 정보영역의 오프셋(Column Information Area Offset)으로, Table 1에서 확인할 수 있다.

Table 1. FRM File Header

Offset	Name	Size (Byte)	Description
0x03	Storage Engine Type	1	스토리지 엔진 타입
0x1E	Record Format	2	레코드 포맷
0x33	MySQL Version	4	MySQL 데이터베이스의 버전
0x44	Column Information Area Offset	2	컬럼 정보 영역 오프셋

3.2.2 키와 인덱스 정보 영역

키와 인덱스 정보 영역(Key, Index information area)은 FRM파일의 0x1000(4,096)의 고정된 오프셋에 위치한다. 해당 영역에서 FRM 파일에 해당하는 테이블에 사용되는 키(Primary Key, Index Key)의 개수와 이에 대한 메타정보가 위치한다.

키의 개수에 대한 정보는 키 영역에서부터 0x0E 오프셋 위치에서 확인할 수 있다. 또한 키 개수에 대한 정보 다음부터 테이블에서 사용되는 키에 대한 메타 정보들이 위치하고 있다. 키 메타정보는 0x09 바이트 길이의 엔트리로 구성되어 있으며, 해당 테이블에서 키 값으로 쓰이는 컬럼 정보가 명시되어 있다.

3.2.3 컬럼 정보 영역(Column information area)

컬럼 정보 영역은 크게 헤더, 컬럼 메타데이터 영역, 컬럼명 영역으로 구성되어 있다. 컬럼 정보 영역의 헤더에서 컬럼 메타데이터 영역의 위치와 컬럼 개수 정보가 존재한다. 컬럼 메타데이터 영역은 Table 2와 같이 17 바이트의 엔트리로 구성되어 있으며, 각 엔트리는 컬럼의 타입, Null 정보, 최대 길이 값 등의 스키마 정보를 저장한다.

Table 2. Column Metadata Entry

Offset	Name	Size (Byte)	Description
0x1000	Sequence Number	1	메타데이터 시퀀스 넘버
0x1001	Column Name Length	1	컬럼의 이름 길이
0x1003	Column Allocation Length	2	컬럼 데이터의 최대 폭
0x1009	Nullable	1	디폴트 NULL 유무
0x100C	Data Type	1	컬럼 데이터 타입
0x100E	Charset Type	1	해당 컬럼의 캐릭터 셋

앞서 언급한 것과 같이 InnoDB에 대한 연구가 활발하게 진행되었지만 아직까지 로우 데이터 상태에서 삭제된 데이터를 레코드 단위로 복구하는 방법은 존재하지 않는다. 또한 MySQL 데이터베이스에서 각 테이블의 스키마 정보가 저장된 FRM 파일에 대한 연구가 이루어 졌지만, InnoDB 데이터베이스에서 삭제된 레코드를 복구하기 위해서는 추가적으로 IBD 파일에 대한 분석이 필요하다.

본 논문에서는 InnoDB 스토리지 엔진에서 생성되는 IBD 파일을 분석하고, 해당 파일에서 사용자가 삭제한 데이터에 대해 레코드 단위로 복구하는 기법에 대해 제안한다.

4. MySQL InnoDB 데이터베이스 구조

MySQL InnoDB 데이터베이스에서 테이블이 생성될 때 스키마 정보를 저장하는 FRM 파일과 사용자 레코드를 저장하는 IBD 파일이 생성된다. InnoDB의 FRM 파일은 앞서 연구된 FRM 파일 구조와 동일하며, IBD 파일 구조는 4.1절에서 자세히 설명한다.

4.1 IBD 파일 구조

IBD 파일은 InnoDB 스토리지 엔진기반의 MySQL 데이터베이스에서 사용자가 저장한 실제 데이터들을 저장한다.

IBD 파일은 0x4000 사이즈의 페이지 단위로 구성되어 있으며, 전체 구조는 Fig. 3과 같다. 각 페이지들은 페이지 번호를 지니고 있으며, 기본적으로 FIL Header와 FIL Trailer 영역이 존재한다. FIL Header 영역은 Fig. 4와 같이 해당 페이지 번호와 페이지 타입에 대한 정보가 존재하며 동일한 레벨을 지닌 페이지들을 가리키는 포인터를 확인할 수 있다.

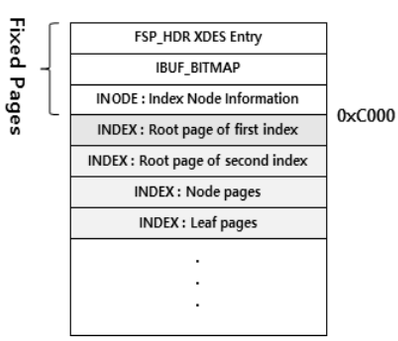

Fig. 3. The Layout of IBD File

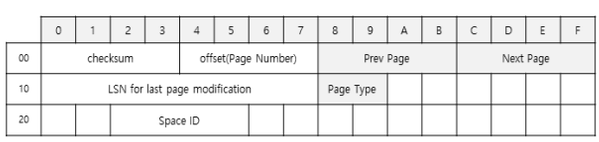

Fig. 4. FIL Header

0번째 페이지부터 2번째 페이지는 IBD 파일의 예약된 영역이며, 실제 사용자가 저장한 데이터는 3번째 페이지(0xC000)

인 Index 페이지를 기준으로 하여 Fig. 5와 같이 B+ 트리 구조로 관리된다. 각 페이지들은 레벨을 지니고 있으며, 실제 사용자 데이터가 저장되는 레코드는 '레벨 0'인 리프 페이지(Leaf Page)에 저장된다. 리프 페이지외의 페이지들은 넌 리프 페이지(None-Leaf Page)라 하며, 리프 페이지들은 넌 리프 페이지의 레코드 내의 Child Page Number를 통해 접근 가능하다.

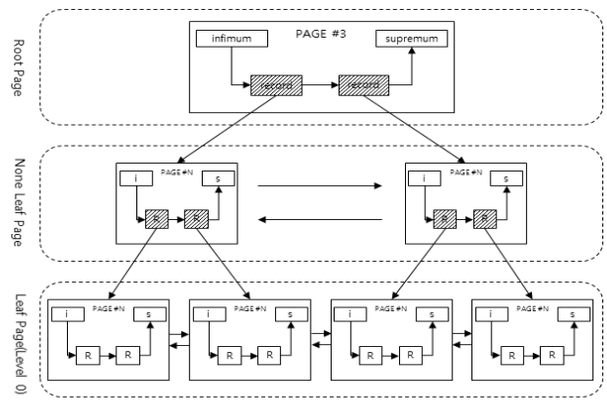

Fig. 5. The B+tree Structure of Index Pages

4.1.1 INDEX 페이지 구조

INDEX 페이지 구조는 Fig. 6과 같이 INDEX 헤더와 시스템 레코드(System Record), 사용자 레코드(User records) 영역으로 구성되어 있다. INDEX 헤더 영역의 구조는 Fig. 7과 같으며, 페이지 레벨, 페이지 내에 저장된 정상레코드의 개수가 있다. 또한, 페이지 내에서 가장 나중에 삭제된 레코드를 가리키는 First Garbage Record Offset 영역을 확인할 수 있다.

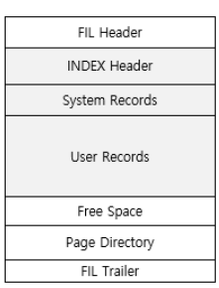

Fig. 6. The Layout of INDEX Page

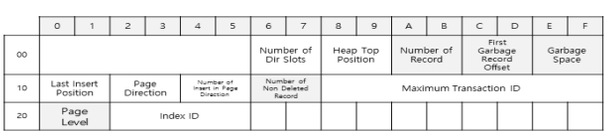

Fig. 7. INDEX Page Header

4.1.2 페이지 레코드

INDEX 페이지의 레코드는 시스템 레코드와 사용자 레코드로 구분된다. 각 레코드는 Fig. 8과 같이 5 바이트 크기의 레코드 헤더를 지니고 있으며, 레코드 헤더의 마지막 2바이트는 해당 레코드 다음 레코드의 위치를 나타낸다.

시스템 레코드는 각 페이지의 고정된 오프셋(0x5E)에 위치하며, infimum record, supermum record가 있다. Fig. 5와 같이 모든 페이지 내의 레코드는 Linked List 형태로 존재한다. 레코드 Linked List의 가장 첫 번째 레코드는 infimum record이고 마지막 레코드는 supermum record가 된다. MySQL 데이터베이스에서 정상레코드는 각 페이지의 시스템 레코드인 infimum record를 시작하여 supermum record를 마지막으로 리스트를 순회하며 추출가능하다.

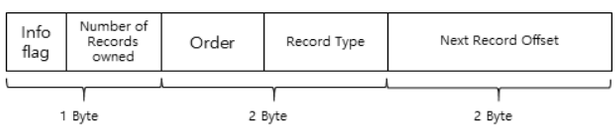

Fig. 8. The Header of Record

4.1.2.1 넌 리프 페이지(None leaf page)의 레코드 구조

넌 리프 페이지의 각 레코드는 키 값을 통해 사용자 데이터가 저장된 리프 페이지를 인덱싱 하여 저장한다. 넌 리프 페이지의 각 레코드의 구조는 Table 3과 같으며, 실제 사용자 데이터가 저장되는 레코드의 페이지는 "Child page number"를 통해 접근 가능하다.

Table 3. Record in None Leaf Page

Offset	Name
-m	Key length information area(Length : m)
-n	Null bitmap area(Length : n)
0	Record header
5	Primary key(Length : k)
5 + k	Child page number

4.1.2.2 리프 페이지(Leaf page)의 레코드 구조

실제 사용자 데이터가 저장되는 리프 페이지의 레코드 구조는 Fig. 9와 같다. 시스템 레코드와 같이 고정된 크기의 레코드 헤더가 있으며, 헤더 앞부분에 각 컬럼의 가변 길이 정보가 저장되는 컬럼 길이 영역과(Column Length Area) 컬럼의 널(NULL) 여부를 나타내는 널 비트맵 영역(Null Bitmap Area)이 위치한다.

널 비트맵 영역과 컬럼 길이 영역은 해당 테이블의 컬럼 개수에 따라 크기가 달라진다. 널 비트맵 영역의 길이는 FRM 파일에서 추출된 컬럼 메타 정보를 통해 길이를 구할 수 있고, 컬럼 길이 영역은 널 비트맵 영역을 참조하여 크기를 구할 수 있다.

레코드 헤더 다음으로는 해당 테이블에서 Primary Key로 사용되는 컬럼의 데이터가 저장되며, Primary Key 값에 대한 데이터 이후에 고정된 크기의 Transaction ID, Rollback Pointer 값이 저장된다.

레코드의 유저 데이터(User data)영역은 Primary Key를 제외한 각 컬럼의 데이터들이 차례대로 저장된다. 문자열과

같은 가변길이를 갖는 데이터 타입을 제외한 Date, 정수 형, 실수 형의 경우 고정길이 형태를 갖으며 바이너리 형태로 저장된다[6]. 이에 대한 자세한 내용은 5절에서 설명한다.

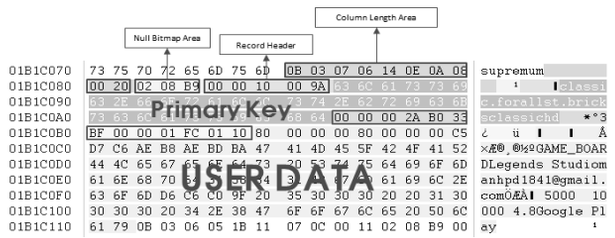

Fig. 9. The Structure of User Record

5. InnoDB 데이터베이스 레코드 단위 복구 기법

MySQL InnoDB Storage Engine의 복구 기법의 전체 알고리즘은 Fig. 10과 같다. 알고리즘은 크게 스키마 구조 복구, IBD파일에서의 삭제된 레코드 복구의 총 두 단계로 구분된다.

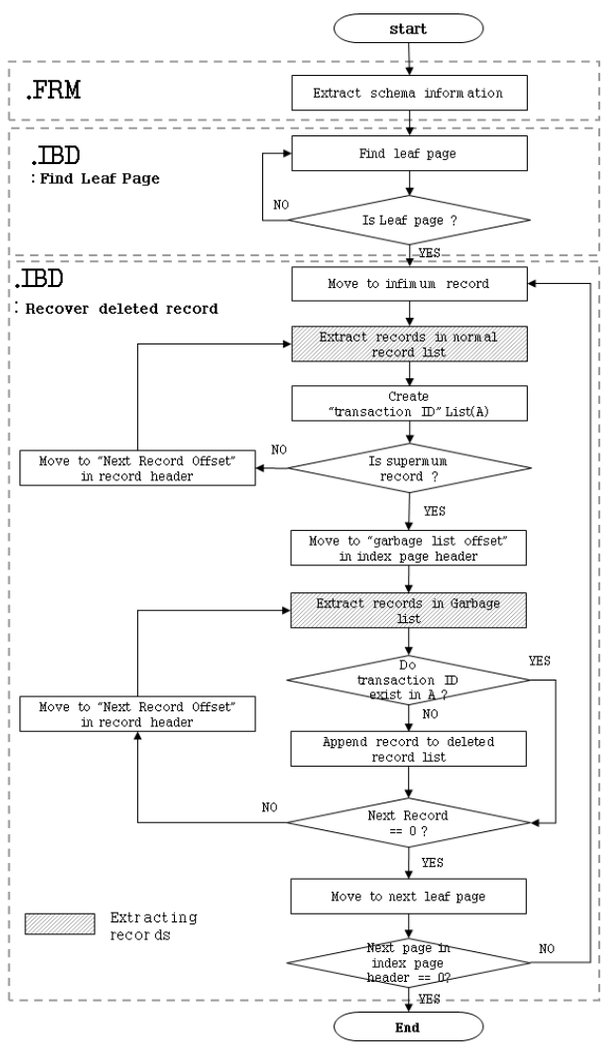

Fig. 10. The Algorithm of Recovering Innodb Database

5.1 FRM 파일에서의 스키마 구조 획득

스키마 구조는 앞서 언급한 것과 같이 FRM 파일을 분석하여 추출한다. FRM 파일의 키 영역에서 키의 개수와 키에 해당하는 컬럼 번호를 추출한다. 컬럼 정보 영역에서는 해당 테이블의 총 컬럼 개수에 대한 정보를 확인한 후, 컬럼 메타데이터를 추출한다. 컬럼 메타데이터는 엔트리 배열로 되어있으며, 앞서 추출한 키 컬럼 번호에 해당하는 컬럼 메타데이터는 해당 테이블의 키 컬럼이라 판단할 수 있다. 컬럼 정보를 모두 추출한 뒤, 레코드 복구 단계로 추출된 컬럼 정보를 전달한다.

5.2 IBD파일에서의 삭제된 레코드 복구

IBD파일 내에서 삭제된 레코드 복구는 사용자 레코드가 저장되는 리프페이지를 탐색, 정상 레코드 복원/삭제 레코드 복구하는 총 두 단계로 구분된다.

5.2.1 리프 페이지 탐색(Find Leaf Page)

실제 사용자 데이터는 모두 리프 페이지에 저장된다. 하지만 데이터가 커질수록 각 페이지들은 4.1절에서 설명한 B+트리 구조를 형성하게 되고, 리프 페이지는 모두 넌 리프 페이지의 레코드의 'Child Page Number'를 통해 인덱싱된다. 동일한 레벨의 페이지들은 FIL Header 포인터를 통해 리스트 형식으로 존재한다(Fig. 5). 따라서 첫 번째 리프 페이지를 탐색할 수 있다면, FIL Header에 위치한 'Next Page Number' 값을 통해 모든 리프 페이지를 순회할 수 있다.

첫 번째 리프 페이지를 찾기 위해 각 넌 리프 페이지의 첫 번째 레코드에서 'Child Page Number'를 통해 페이지를 이동하고, 이동한 페이지 레벨이 "0"이 될 때까지 탐색한다. 이에 대한 알고리즘은 Fig. 11과 같다.

```
int FindLeafPage(Page start offset)
    Page Level = Page start offset + 0x41;
    if Page Level == 0 then
        return Page start offset;
    end
    else then
        next offset = infimum record header -> Next record offset;
        first non-clustered record = infimum record offset + next offset;
        child page number = first non-clustered record -> child page number;
        Page start offset = child page number * 0x4000;
        return FindLeafPage(Page start offset);
    end
end
```

Fig. 11. The Algorithm of Searching Index Page

5.2.2 정상 레코드 복원/삭제된 레코드 복구(Recover deleted record)

각 레코드 추출 순서도는 Fig. 12와 같이 Extract column null information, Extract length of columns, Extract primary key, Extract transaction ID/Rollback pointer, Extract user data의 5단계로 나눠진다. 각 단계마다 레코드를 추출하기 위

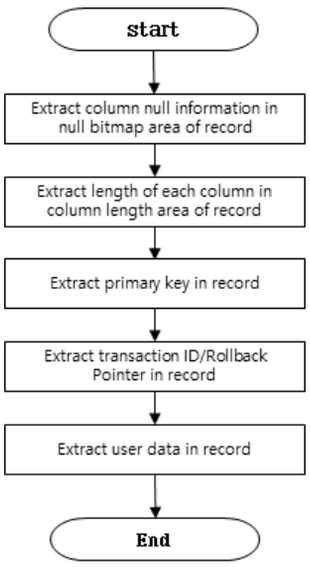

Fig. 12. Extracting Records in Fig. 10.

```
Number of column ;
Null bitmap size ;
Column[] <- column information in FRM files ;
Null bitmap[] <- null bitmap information ;
Record <- current record ;

Index = 1 ;
for I = 0 ; I < Null bitmap size ; I++ :
    shift register = 0x01 ;
    bit index = 0 ;
    while bit index < 8 do :
        if Index == Number of column then :
            break ;
        end
        if Column[Index] -> metadata -> Not Null Bit & Not Null == 0 then :
            Record -> COLUMN[Index++] -> is Null = False ;
            Continue ;
        end
        Record -> COLUMN[Index] -> is Null = Null bitmap[I] & shift register ;
        shift register = shift register << 1 ;
        Index++ ;
        bit_index++;
    end
end
```

Fig. 13. The Algorithm of Extracting Null Information

해서 5.1절에서 추출된 컬럼 정보에 대한 데이터를 활용하며, 레코드 추출 절차는 아래와 같다.

STEP 1. (Extract column null information in Null bitmap area) 4.1.2.1절에서 언급하였듯이 사용자 레코드 전체 길이는 가변 길이로 되어 있다. 레코드 전체 길이를 구하기 위해 가장 먼저 널 비트맵 영역(Null Bitmap Area)의 길이와 널 비트맵 영역을 통해 각 컬럼의 널(NULL)값에 대한 정보를 구해야 한다. 각 컬럼이 널 값을 가질 수 있는지에 대한 정보는 앞서 추출한 컬럼 메타데이터에서 확인할 수 있으며, 키 값에 해당하는 컬럼은 널 값을 가질 수 없기 때문에 널 비트맵영역에서 제외된다. 키 값을 제외한 모든 컬럼의 메타데이터를 확인하여 널 비트맵영역의 크기(N)를 구하고 각 컬럼이 널 값을 갖는지에 대한 정보를 추출해야 한다. 널 비트맵 영역의 길이를 구하는 식은 다음 Equation (1)과 같으며, 널 값의 유무에 대한 정보를 획득하는 알고리즘은 Fig. 13과 같다.

$$\text{Null Bitmap Area Length} = (N \,\%\, 8 == 0)\ ?\ N / 8 : N / 8 + 1 \quad (1)$$

STEP 2. (Extract legth of each column in column length area) 각 컬럼의 널 값에 대한 정보를 추출한 뒤, 해당 정보를 참조하여 컬럼 길이 정보 영역에서 각 컬럼의 길이 정보를 추출해야 한다. 레코드의 컬럼 길이 정보영역에서 각 컬럼의 길이는 널(NULL)값을 갖지 않는 컬럼 중, 가장 첫 번째 컬럼에 대한 길이 정보부터 나열된다. 만약 특정 컬럼이 널 값을 가지게 되면 컬럼 길이에 대한 정보는 생략된다. 그러므로 앞서 추출한 널 값에 대한 정보를 통해 컬럼이 널 값을 갖는다면 컬럼 길이는 널(NULL) 값이라 할 수 있고 널 값을 갖지 않는다면 컬럼 길이 정보 영역에서 해당 컬럼의 길이를 추출한다.

STEP 3. (Extract primary key) 각 컬럼 길이 정보를 추출한 뒤, 추출된 길이 정보를 기반으로 Primary Key 영역의 데이터를 추출한다.

STEP 4. (Extract transaction ID and Rollback pointer) 레코드의 트랜잭션 아이디(Transaction ID)와 롤백포인터(Rollback Pointer)는 각각 6바이트, 7바이트로 고정길이로 되어있다. 추출된 트랜잭션 아이디는 삭제 레코드 복구에 활용되므로, 리스트에 저장한다.

STEP 5. (Extract user data) 앞서 추출된 컬럼 정보와 각 컬럼 길이 정보를 참조하여 사용자가 저장한 데이터를 추출한다. 컬럼 정보를 통해 각 레코드 내의 컬럼 타입을 구분하고, 컬럼타입을 구분한 뒤 해당 컬럼이 가변길이를 갖는 컬럼인 경우 추출된 컬럼 길이 정보를 통해 사용자가 저장한 컬럼 데이터를 확인할 수 있다.

5.2.2.1 정상 레코드 복원

각 리프 페이지에 저장된 정상 레코드의 수는 인덱스 헤더영역에서 확인할 수 있으며, 정상 레코드는 각 리프 페이지의 시스템 레코드인 infimum record에서부터 supermum record 까지 리스트 순회하며 추출할 수 있다.

정상 레코드를 추출하면서 삭제된 레코드를 복구할 때 사용할 각 레코드의 트랜잭션 ID에 대한 리스트를 생성한다. 이에 대한 자세한 설명은 5.2.2.2절에서 한다.

5.2.2.2 삭제된 레코드 복구

삭제된 레코드들도 정상 레코드와 마찬가지로 각 레코드의 헤더 영역의 'Next Record Offset' 값을 통해 삭제된 레코드들끼리 리스트를 형성한다.

각 리프 페이지 인덱스 헤더의 'First Garbage Offset'을 통해 삭제된 레코드들에 접근할 수 있다. First Garbage

Offset에서 가리키는 곳은 가장 마지막으로 삭제된 레코드의 키 영역을 가리킨다. 각 리프 페이지에서는 해당 오프셋을 통해 해당 리프 페이지에서 마지막으로 삭제된 레코드에 접근한 후, 레코드 추출 알고리즘을 적용하여 삭제된 레코드들을 모두 복구한다.

각 레코드의 트랜잭션 아이디는 테이블에서 생성되는 모든 레코드들의 고유 값을 나타낸다. 각 페이지의 'First Garbage Offset'은 페이지가 이동되었을 시 삭제된 레코드들도 포함된다. 페이지가 이동되어 삭제된 레코드들은 사용자가 직접 삭제하지 않은 레코드로써, 실제 삭제된 레코드와 앞서 생성된 트랜잭션 아이디 리스트를 통해 구분할 수 있다. 'First Garbage Offset'을 순회하면서 삭제된 레코드를 복구한 후, 트랜잭션 ID 리스트에 삭제된 레코드에 대한 아이디가 존재한다면, 복구된 레코드는 페이지 이동시 삭제된 '정상 레코드'라 판단할 수 있다. 반대로 리스트에 복구된 아이디가 존재하지 않는다면 해당 레코드는 실제로 사용자가 삭제한 레코드라고 판단할 수 있다.

6. 실험 결과

본 논문에서 제안된 알고리즘을 기반으로 제작된 도구를 통해 복구 율에 대한 실험을 실시하였다. 실험용 데이터베이스는 실제 MySQL InnoDB 데이터베이스와 관련된 사건에서 수집된 자료 형으로 구성하였으며, 데이터베이스의 테이블 스키마는 Table 4와 같다.

Table 4. Table Schema Information

Case Number	Column Name	Data Type
CASE 1	idx	int(10)
	class_idx	int(10
	user_idx	int(10)
	book_idx	int(10)
	book_contents_idx	int(10)
	title	varchar(1024)
CASE 2	idx	int(10)
	begin	timestamp
	end	timestamp
	data	longtext
CASE 3	SHA256	varchar(64)
	PackageName	varchar(255)
	AppName	tinytext
	Category	tinytext
	Developer	tinytext
	AppvVersionCode	tinytext

각 케이스마다 총 200개의 레코드를 생성하였으며, 50개의 데이터를 임의로 삭제하였다. 그리고 다시 50개의 새로운 데이터를 추가한 후, 50개의 데이터를 삭제하였다. 제안된 알고리즘으로 제작된 도구를 통해 복구한 결과는 Table 5와 같다.

Table 5. InnoDB Recovery Result

Case Number	Number of deleted record	Number of recovered record	Recovery rate
CASE 1	100	46	46%
CASE 2	100	47	47%
CASE 3	100	43	43%
Average rate			45%

리프 페이지는 B+ Tree 구조로 구성되어 있으며, 레코드들이 추가되거나 삭제될 시 페이지 이동으로 인해 삭제된 레코드가 덮어써질 수 있다. 처음 50개 삭제된 데이터들은 향후 추가된 50개의 데이터들과 레코드의 길이가 완전히 일치하여 덮어써져 복구하지 못하였다. 하지만 이 후 삭제된 50개의 데이터들을 포함한 덮어써지지 않은 삭제된 레코드는 제안된 알고리즘으로 모두 정상적으로 복구가 가능한 것을 확인할 수 있었다.

위와 같은 실험을 통해 MySQL InnoDB 데이터베이스에서 사용자 데이터가 IBD 파일에 온전히 남아있다면, 레코드 단위로 복구가 가능한 것을 입증하였다.

7. 결 론

MySQL InnoDB 데이터베이스는 많은 기관, 기업에서 주요 정보와 기록들을 저장하고 관리하는 목적으로 사용되고 있으므로 삭제된 데이터 복구에 대한 연구는 포렌식 관점에서 중요하다.

이에 본 논문은 InnoDB 스토리지 기반의 MySQL 데이터베이스에서 생성되는 FRM과 IBD 파일 포맷을 상세히 분석하여, FRM 파일에서는 각 테이블의 스키마 구조를 파싱하고 IBD 파일에서는 실제 삭제된 레코드를 추적하여 이를 레코드 단위로 복구할 수 있는 알고리즘에 대해 제시하였다.

제안된 알고리즘을 기반으로 레코드 복구 도구를 제작하였으며, 해당 도구로 실험한 결과 삭제된 레코드를 정상적으로 복구하는 것을 입증하였다. 본 논문에서 제안한 MySQL InnoDB에 대한 레코드 복구 기법은 실제 수사에서 큰 도움이 될 것으로 기대된다.

References

[1] K. S. Lim, D. C. Lee, J. H. Park, and S. J. Lee, "A Novel Database Forensic Technique Using Table Relationship Analysis," *Korea Multimedia Society Fall Conference Proceedings*, pp.65-68, 2009.

[2] D. C. Lee and S. J. Lee, "Research of organized data extraction method for digital investigation in relational detabase system," *Journal of the Korea Institute of Information Security and Cryptology*, Vol.22, No.3, PP.565-573, 2012.

[3] P. Fruhwirt and M. Huber, "InnoDB database forensics," *24th IEEE International Conference on Advanced Information Net Working and Applications (AINA)*. pp.1028-1036, Apr., 2010 .

[4] P. Fruhwirt, P. Kieseberg, S. Schrittwieser, M. Huber, and E. Weippl, “InnoDB database forensics: Reconstructing data mani pulation queries from redo logs,” *Seventh International Conference on Availability, Reliability and Security (ARES)*, pp.625-633, Aug., 2012

[5] W. S. Noh, S. M. Jang, C. H. Kang, K. M. Lee, and S. J. Lee, “The Method of Deleted Record Recovery for MySQL MyISAM Database,” *Journal of the Korea Institute of Information Security and Cryptology*, Vol.26, No.1, pp.125-134, 2016.

[6] Paul DuBois, MySQL Developer’s Library, 4th Ed., Addison-Wesley Professional, Aug., 2008.

[7] Oracle, SHOW COLLATION Syntax [Internet], htt ps://dev.mysql.com/doc/refman/5.0/en /show-collation.html.

장 지 원

e-mail : jeewon0838@gmail.com
2015년 국민대학교 컴퓨터공학(학사)
2015년～현 재 고려대학교 정보보호대학원 정보보호학과 석사과정
관심분야 : Digital Forensic & Information Security

정 두 원

e-mail : dwjung77@korea.ac.kr
2011년 고려대학교 산업경영공학과(학사)
2011년～현 재 고려대학교 정보보호대학원 정보보호학과 석·박사통합과정
관심분야 : Digital Forensic, Information Security, Big Data Analysis

이 상 진

e-mail : sangjin@korea.ac.kr
1987년 고려대학교 수학과(학사)
1989년 고려대학교 수학과(석사)
1994년 고려대학교 수학과(박사)
1989년～1999년 ETRI 선임연구원
1999년～현 재 고려대학교 정보보호대학원 교수
2008년～현 재 고려대학교 디지털포렌식연구센터 센터장
관심분야 : Digital Forensic, Steganography, Hash Function