Research project
MySQL record carving

Esan Wit   Leendert van Duijn
Supervisor: Kevin Jonkers

February 10, 2014

Abstract

Databases are the driving force for many applications. These databases may contain sensitive or mission critical information. When records are deleted remnants of data remain on the filesystem. This project describes a method for recovering data from MySQL databases after deletion. It discusses the use of template matching and data validation to find records which may have been deleted. The proof of concept implementation can successfully retrieve parts of the deleted data. The methodology described can also be used on other systems which store data in similar, structured fashion.

Contents

1 Introduction 1

2 Related work 2

3 Goal 3

4 Background 4
- 4.1 MyISAM 4
  - 4.1.1 Row formats 4
- 4.2 InnoDB 5

5 Observations 6
- 5.1 MyISAM static 6
- 5.2 MyISAM dynamic 8
- 5.3 InnoDB Antelope Compact 9
- 5.4 InnoDB Antelope Redundant 11

6 Methods 13
- 6.1 Template Matching 13
  - 6.1.1 Datatypes 13
  - 6.1.2 Caveats 13
  - 6.1.3 Consideration of NULL fields 14
  - 6.1.4 Consideration of record headers 14
- 6.2 Validation 14
  - 6.2.1 Field validators 15
  - 6.2.2 Row validators 15
  - 6.2.3 Example validators 16
- 6.3 Scanning 16
  - 6.3.1 Consideration of data of interest 16
- 6.4 High level algorithm 17

7 Proof of Concept 18

8 Tests 19
- 8.1 Small scale, InnoDB Compact with deleted entries 19
- 8.2 World dataset 20
  - 8.2.1 MyISAM 21
  - 8.2.2 InnoDB 22

9 Conclusion 25

10 Further research 26

A Documentation 29
A.1 Scanner usage and configuration . . . 29
A.1.1 Settings . . . 30
A.1.2 Validators . . . 30
A.2 Generic field types . . . 32
A.2.1 Null field . . . 32
A.2.2 Non specific field . . . 32

B Documentation MyISAM engine 32
B.1 Headers . . . 33
B.1.1 Dynamic row format . . . 33
B.2 Scanner supported datatypes . . . 34
B.2.1 Blobs . . . 34
B.2.2 CharFixed . . . 35
B.2.3 VarChar . . . 35
B.2.4 Texts . . . 35
B.2.5 Float, Double and Real . . . 36
B.2.6 Decimal and Numeric . . . 36
B.2.7 Integers . . . 37
B.2.8 Datetime . . . 37
B.2.9 Date . . . 37
B.2.10 Time . . . 38
B.2.11 Timestamp . . . 38
B.2.12 Year . . . 38
B.2.13 Enum . . . 38
B.2.14 Set . . . 39

C Documentation InnoDB engine 39
C.1 Row format . . . 39
C.1.1 Primary key . . . 39
C.1.2 Transaction ID and Roll pointer . . . 39
C.1.3 Redundant header and Record Directory . . . 40
C.1.4 Compact header and variable length header . . . 40
C.2 Scanner supported types . . . 41
C.2.1 Nullable fields . . . 43
C.2.2 CharFixed . . . 43
C.2.3 VarChar . . . 43
C.2.4 Integers . . . 43
C.2.5 Bit . . . 43

C.2.6 Set 43
C.2.7 Float Double and Real 43
C.2.8 Date 44
C.2.9 Time 44
C.2.10 Timestamp 44
C.2.11 Year 44
C.2.12 Enum 44
C.2.13 Decimal and Numeric 44
C.2.14 Datetime 44

D Configuration examples 44

E Data used in tests 46
E.1 Deletion and header structure 46
E.2 Simple InnoDB compact test 48
E.3 World dataset 49
E.3.1 MyISAM configuration 50
E.3.2 InnoDB configuration 52

1 Introduction

MySQL is a popular database implementation and is commonly used in many LAMP-based applications. Upon deletion of database record by means of a `DELETE` statement the data is considered removed. MySQL databases, and also other database systems[1], do not remove or overwrite the information on disk. As a result parts of the data are recoverable which may not be intended.

When searching vendor websites on how to recover deleted records developers say that this data is inaccessible or requires extensive effort to partially restore[1]. Although 7 years have passed, there are not many readily available tools which attempt to extract this data. Tools that do exist suffer from a high false positive rate[2, 3].This research tries to find a method of data recovery which can be used on database systems to recover the data remaining after deletion, and is generic enough to be usable for different database systems.

Forensic research into databases, and particularly record carving, is not as developed as forensic research into file carving but many techniques are compatible between these fields. Most research pertaining to database systems is focussed on using metadata or recovery using alternative sources[1, 4, 5, 6, 7]. In contrast file carving is a well established field and the file carving toolkit is an ever-expanding one. This research intends to bring the two fields closer by using techniques developed for carving entire files to extend methods for carving database records. It is a continuation of research done into record recovery on SQLite databases by Pooters et al.[2] and applying those techniques on MySQL. The proposed method extends the earlier research by using validation techniques to reduce invalid results. Data validation of carved records is important to enable data carving of partial data, which may have been left behind, without having to manually filter through an abundance of false positives.

[1] `http://forums.mysql.com/read.php?21,135990,137776#msg-137776`

2 Related work

Database systems are not typically designed with any cleanup when it comes to deleted records. Work by Stahlberg et al.[1] shows that many database systems are prone to leaving data behind, at least partially, upon deletion.

Some research has been done in analyzing different aspects of database systems. Kieseberg et al.[7] proposes a method for validating the data contained in the database and proving authenticity. Frühwirt et al.[5] focusses on recovery of manipulation queries, `UPDATE`, `INSERT` and `DELETE`, from log files. But less research is done in recovery of the deleted records.

Existing solutions proposed by Pooters et al.[2] and Percona LLC[3] are based on template matching to recover data of specific databases. These solutions, however, are targeted to very specific database systems and both suffer from high false positive rates for simple templates.

Garfinkel[8] proposes the use of validation for carving fragmented files. Although this technique is primarily meant for validating fragmented files the principle can be used in template matching to reduce false positives incurred by template matching.

3 Goal

The goal of this project is to develop a carving method for the recovery of deleted database records that works for MySQL, version 5.6.16 which is the currently recommended release. For this we must first investigate the following:

- What data remains after deletion of a record?
- What methods exist for recovering (parts of) this data?
- Can this be extended to recover from more general damage[2] or other database systems?
- How do the differences between competing database systems relate to record recovery?

The answers to these questions will be used to create a proof of concept implementation to recover deleted records.

[2]For instance file system corruption, incorrect or missing indexes

4 Background

First, an understanding of the MySQL internals is required to decide on how to proceed. Since MySQL is designed to allow data to be handled by different storage engines[9, chapter 14] there is a significant difference in how data is stored on disk. The storage engine is a module determining how queries should be executed, data stored, indexes maintained and more. Each storage engine can therefore employ vastly different techniques for handling data.

This project focusses on InnoDB and MyISAM. These are the two most common storage engines used by MySQL installations. MyISAM was the previous default storage engine and InnoDB is the new default storage engine since MySQL version 5.5[3]. As such these engines are considered most relevant.

4.1 MyISAM

MyISAM uses an Indexed Sequential Access Method(ISAM) file overlay to store all rows. A row holds both the record data and record header, also called a record. All data is stored in a `.MYD` file. This file contains all rows in a sequential, but possibly unsorted, order. If a record is deleted it is marked as deleted on the position it is located. A file could then possibly look like figure 1.

<table>
<tr><td colspan="3">Record 1</td><td>Record 2</td><td>Record</td></tr>
<tr><td>3</td><td>Record 4</td><td colspan="3">Record 5</td></tr>
<tr><td colspan="3">Deleted record</td><td colspan="2">Record 7</td></tr>
<tr><td colspan="5">...</td></tr>
</table>

Figure 1: Layout of a .MYD file

4.1.1 Row formats

MyISAM comes with two flavors for the storage. A fixed row format and a dynamic row format. Each row format is meant to improve on either storage requirements of data or access time of data.

Fixed row format In the fixed row format each row uses a record of identical length to store data. Regardless of the length of actual data present in the record. For instance strings which are stored may be padded with

[3] `http://dev.mysql.com/doc/refman/5.5/en/innodb-default-se.html`

spaces to fill the required length. This enables the engine to efficiently read the record from disk.

Dynamic row format The dynamic row format stores a record in as much space as is required to hold the data. This is the default row format when there is a column of variable length in the table, e.g. text, varchar or blob. A text column containing a string of length n can take as little as n bytes storage, see section B.2.4 for details.

4.2 InnoDB

The InnoDB engine is the new default engine used by MySQL, as opposed to MyISAM it uses a hierarchical page based storage file.

InnoDB has two file formats `Antelope` and `Barracuda`, the latter supports two more row types which change the storage type for `TEXT` and `BLOB` fields.

Data is stored in pages, with a maximum size of 16KB[10, section 21.2]. These pages are organized in a B-tree index, with the so called `clustered index` containing the actual records. The clustered index is stored in the `ibd` file.

This research is focused on the so called leaf pages which hold the record data including a header. Due to the limited page size some data or column types are not stored locally but on a separate, `overflow` page, as opposed to the purely sequential storage of MyISAM data files. The overflow pages are separate from the leaf pages and are reserved for larger objects This means that any carver should implement functionality to carve the linked storage page. Also if the linkage data is overwritten or lost it becomes impossible to match the data to the original record by conventional means.

In the older row formats (see C.1) larger fields are stored partially off page, with the first 768 bytes locally stored together with a 20 byte value to locate the remainder[9, Section 14.2.9.4]. The newer formats store only the 20 byte value to the full field stored off page[9, Section 14.2.9.3], saving space in the page where the record is stored.

5 Observations

Our experiments show several properties which can be instrumental in recovering records after deletion.

While a DELETE statement is designed to remove a record from the database, neither InnoDB or MyISAM overwrite deleted records in place. The header data is updated to reflect the new state of the record which may also overwrite data from the beginning of the record. Our experiments show deletion is handled differently depending on the storage engine and selected row format.

When deleting records we observed that though deleted their data remained until they were overwritten by further statements. Small scale testing on InnoDB suggests that on deletion all records remain partially intact until we delete the last record at which point the software reinitialized parts of the datafile. The same test on MyISAM revealed that even though the last record in the datafile was deleted, it was not directly truncated and could be partially recovered.

In order to recover a deleted record, and distinguish between intact records, we analyzed deletion of records for several distinct row formats. The tests were done on several records and tables, giving consistent results regardless which specific record was deleted.

5.1 MyISAM static

Under the MyISAM Static row format parts of the data remain after deletion, this can be observed in figure2 where part of the data file is shown before and after deletion. The underlined bytes in the figure are changed due to the deletion and the ~~crossed~~ bytes belong to a field which is no longer complete and thus makes a complete recovery of this data impossible.

Due to the fixed size of fields in the fixed row format deletion does not prevent recovery of data beyond the overwritten header and fields, as both sizes and relative offsets in a record are constant. Thus deletion prevents full recovery of data, but does not prevent partial recovery of longer rows.

The records are color coded to show the individual fields, the schema for this table can be found in E.1

```
0070   66 38 33 63 39 64 00 00 00 00 00 00 00 00 00 00   |f83c9d..........|
0080   52 ea 57 a3 00 00 00 00 00 fd 02 00 00 00 12 4c   |R.W............L|
0090   65 65 6e 64 65 72 74 20 76 61 6e 20 44 75 69 6a   |eendert van Duij|
00a0   6e 00 00 18 4c 65 65 6e 64 65 72 74 2e 76 61 6e   |n...Leendert.van|
00b0   44 75 69 6a 6e 40 6f 73 33 2e 6e 6c 00 00 00 00   |Duijn@os3.nl....|
00c0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   |................|
00d0   00 00 00 00 00 00 28 35 62 61 61 36 31 65 34 63   |......(5baa61e4c|
00e0   39 62 39 33 66 33 66 30 36 38 32 32 35 30 62 36   |9b93f3f0682250b6|
00f0   63 66 38 33 33 31 62 37 65 65 36 38 66 64 38 00   |cf8331b7ee68fd8.|
0100   00 00 00 00 00 00 00 00 00 52 ea 57 a3 00 00 00   |.........R.W....|
0110   00 00 fd 03 00 00 00 0d 4b 65 76 69 6e 20 4a 6f   |........Kevin Jo|
0120   6e 6b 65 72 73 00 00 00 00 00 00 00 11 6a 6f 6e   |nkers........jon|
```

Original

```
0070   66 38 33 63 39 64 00 00 00 00 00 00 00 00 00 00   |f83c9d..........|
0080   52 ea 57 a3 00 00 00 00 00 00 ff ff ff ff ff ff   |R.W.............|
0090   65 65 6e 64 65 72 74 20 76 61 6e 20 44 75 69 6a   |eendert van Duij|
00a0   6e 00 00 18 4c 65 65 6e 64 65 72 74 2e 76 61 6e   |n...Leendert.van|
00b0   44 75 69 6a 6e 40 6f 73 33 2e 6e 6c 00 00 00 00   |Duijn@os3.nl....|
00c0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   |................|
00d0   00 00 00 00 00 00 28 35 62 61 61 36 31 65 34 63   |......(5baa61e4c|
00e0   39 62 39 33 66 33 66 30 36 38 32 32 35 30 62 36   |9b93f3f0682250b6|
00f0   63 66 38 33 33 31 62 37 65 65 36 38 66 64 38 00   |cf8331b7ee68fd8.|
0100   00 00 00 00 00 00 00 00 00 52 ea 57 a3 00 00 00   |.........R.W....|
0110   00 00 fd 03 00 00 00 0d 4b 65 76 69 6e 20 4a 6f   |........Kevin Jo|
0120   6e 6b 65 72 73 00 00 00 00 00 00 00 11 6a 6f 6e   |nkers........jon|
```

After deletion

Figure 2: Fragment of MyISAM data file in Fixed row format

5.2 MyISAM dynamic

Under the MyISAM Dynamic row format parts of the data remain on deletion, this can be observed in figure 3 where part of the data file is shown before and after deletion. The underlined bytes in the figure are changed due to the deletion, the ~~crossed~~ bytes belong to a field which is no longer complete and thus makes a complete recovery of this data impossible.

The records are color coded to show the individual fields, the schema for this table can be found in E.1

```
040  30 61 30 34 61 64 66 38 33 63 39 64 52 ea 57 a3  |0a04adf83c9dR.W.|
050  00 00 00 00 03 00 63 01 08 fe 02 00 00 00 12 4c  |......c.......L|
060  65 65 6e 64 65 72 74 20 76 61 6e 20 44 75 69 6a  |eendert van Duij|
070  6e 18 4c 65 65 6e 64 65 72 74 2e 76 61 6e 44 75  |n.Leendert.vanDu|
080  69 6a 6e 40 6f 73 33 2e 6e 6c 28 35 62 61 61 36  |ijn@os3.nl(5baa6|
090  31 65 34 63 39 62 39 33 66 33 66 30 36 38 32 32  |1e4c9b93f3f06822|
0a0  35 30 62 36 63 66 38 33 33 31 62 37 65 65 36 38  |50b6cf8331b7ee68|
0b0  66 64 38 52 ea 57 a3 00 00 00 00 00 03 00 57 01  |fd8R.W........W.|
0c0  08 fe 03 00 00 00 0d 4b 65 76 69 6e 20 4a 6f 6e  |.......Kevin Jon|
```

Original

```
040  30 61 30 34 61 64 66 38 33 63 39 64 52 ea 57 a3  |0a04adf83c9dR.W.|
050  00 00 00 00 00 00 00 68 ff ff ff ff ff ff ff ff  |.......h........|
060  ff ff ff ff ff ff ff ff 76 61 6e 20 44 75 69 6a  |........van Duij|
070  6e 18 4c 65 65 6e 64 65 72 74 2e 76 61 6e 44 75  |n.Leendert.vanDu|
080  69 6a 6e 40 6f 73 33 2e 6e 6c 28 35 62 61 61 36  |ijn@os3.nl(5baa6|
090  31 65 34 63 39 62 39 33 66 33 66 30 36 38 32 32  |1e4c9b93f3f06822|
0a0  35 30 62 36 63 66 38 33 33 31 62 37 65 65 36 38  |50b6cf8331b7ee68|
0b0  66 64 38 52 ea 57 a3 00 00 00 00 00 03 00 57 01  |fd8R.W........W.|
0c0  08 fe 03 00 00 00 0d 4b 65 76 69 6e 20 4a 6f 6e  |.......Kevin Jon|
```

After deletion

Figure 3: Fragment of MyISAM data file in Dynamic row format

5.3 InnoDB Antelope Compact

Under the InnoDB Compact row format both the data and significant parts of the header (documented in C.1.4) remain intact on deletion, this can be observed in figure4 where part of the data file is shown before and after deletion. The underlined bytes in the figure are changed due to the deletion.

The changed fields are reserved for the Transaction ID and Roll pointer (see C.1.2) are overwritten and including parts of the header so it represents a correct state after deletion .

The records are color coded to show the individual fields, the schema for this table can be found in E.1

```
0c090  01 10 45 73 61 6e 20 57 69 74 45 73 61 6e 2e 57  |..Esan WitEsan.W|
0c0a0  69 74 40 6f 73 33 2e 6e 6c 38 62 65 33 63 39 34  |it@os3.nl8be3c94|
0c0b0  33 62 31 36 30 39 66 66 66 62 66 63 35 31 61 61  |3b1609fffbfc51aa|
0c0c0  64 36 36 36 64 30 61 30 34 61 64 66 38 33 63 39  |d666d0a04adf83c9|
0c0d0  64 52 ea 57 a3 00 00 00 00 80 28 18 12 00 00 00  |dR.W......(.....|
0c0e0  18 00 75 00 00 00 02 00 00 00 01 81 4e b9 00 00  |..u.........N...|
0c0f0  01 66 01 1d 4c 65 65 6e 64 65 72 74 20 76 61 6e  |.f..Leendert van|
0c100  20 44 75 69 6a 6e 4c 65 65 6e 64 65 72 74 2e 76  | DuijnLeendert.v|
0c110  61 6e 44 75 69 6a 6e 40 6f 73 33 2e 6e 6c 35 62  |anDuijn@os3.nl5b|
0c120  61 61 36 31 65 34 63 39 62 39 33 66 33 66 30 36  |aa61e4c9b93f3f06|
0c130  38 32 32 35 30 62 36 63 66 38 33 33 31 62 37 65  |82250b6cf8331b7e|
0c140  65 36 38 66 64 38 52 ea 57 a3 00 00 00 00 80 28  |e68fd8R.W......(|
0c150  11 0d 00 00 00 20 ff 18 00 00 00 03 00 00 00 01  |..... ..........|
0c160  81 4e b9 00 00 01 66 01 2a 4b 65 76 69 6e 20 4a  |.N....f.*Kevin J|
```

Original

```
0c090  01 10 45 73 61 6e 20 57 69 74 45 73 61 6e 2e 57  |..Esan WitEsan.W|
0c0a0  69 74 40 6f 73 33 2e 6e 6c 38 62 65 33 63 39 34  |it@os3.nl8be3c94|
0c0b0  33 62 31 36 30 39 66 66 66 62 66 63 35 31 61 61  |3b1609fffbfc51aa|
0c0c0  64 36 36 36 64 30 61 30 34 61 64 66 38 33 63 39  |d666d0a04adf83c9|
0c0d0  64 52 ea 57 a3 00 00 00 00 80 28 18 12 00 20 00  |dR.W......(... .|
0c0e0  18 00 00 00 00 00 02 00 00 00 01 81 74 54 00 00  |............tT..|
0c0f0  02 27 06 45 4c 65 65 6e 64 65 72 74 20 76 61 6e  |.'.ELeendert van|
0c100  20 44 75 69 6a 6e 4c 65 65 6e 64 65 72 74 2e 76  | DuijnLeendert.v|
0c110  61 6e 44 75 69 6a 6e 40 6f 73 33 2e 6e 6c 35 62  |anDuijn@os3.nl5b|
0c120  61 61 36 31 65 34 63 39 62 39 33 66 33 66 30 36  |aa61e4c9b93f3f06|
0c130  38 32 32 35 30 62 36 63 66 38 33 33 31 62 37 65  |82250b6cf8331b7e|
0c140  65 36 38 66 64 38 52 ea 57 a3 00 00 00 00 80 28  |e68fd8R.W......(|
0c150  11 0d 00 00 00 20 ff 18 00 00 00 03 00 00 00 01  |..... ..........|
0c160  81 4e b9 00 00 01 66 01 2a 4b 65 76 69 6e 20 4a  |.N....f.*Kevin J|
```

After deletion

Figure 4: Fragment of InnoDB data file in COMPACT row format

5.4 InnoDB Antelope Redundant

Under the InnoDB Redundant row format both the data and significant parts of the header (documented in C.1.3) remain intact on deletion, this can be observed in figure 5 where part of the data file is shown before and after deletion. the underlined bytes in the figure are changed due to the deletion.

The changed fields are reserved for the Transaction ID and Roll pointer (see C.1.2) are overwritten and including parts of the header so it represents a correct state after deletion .

The records are color coded to show the individual fields, the schema for this table can be found in E.1

```
0c090   00 00 00 01 81 67 cb 00 00 01 73 01 10 45 73 61   |.....g....s..Esa|
0c0a0   6e 20 57 69 74 45 73 61 6e 2e 57 69 74 40 6f 73   |n WitEsan.Wit@os|
0c0b0   33 2e 6e 6c 38 62 65 33 63 39 34 33 62 31 36 30   |3.nl8be3c943b160|
0c0c0   39 66 66 66 62 66 63 35 31 61 61 64 36 36 36 64   |9fffbfc51aad666d|
0c0d0   30 61 30 34 61 64 66 38 33 63 39 64 52 ea 57 a3   |0a04adf83c9dR.W.|
0c0e0   00 00 00 00 80 6c 6b 67 63 3b 23 11 0a 04 00 00   |.....lkgc;#.....|
0c0f0   18 13 01 6f 00 00 00 02 00 00 00 01 81 67 cb 00   |...o.........g..|
0c100   00 01 73 01 1d 4c 65 65 6e 64 65 72 74 20 76 61   |..s..Leendert va|
0c110   6e 20 44 75 69 6a 6e 4c 65 65 6e 64 65 72 74 2e   |n DuijnLeendert.|
0c120   76 61 6e 44 75 69 6a 6e 40 6f 73 33 2e 6e 6c 35   |vanDuijn@os3.nl5|
0c130   62 61 61 36 31 65 34 63 39 62 39 33 66 33 66 30   |baa61e4c9b93f3f0|
0c140   36 38 32 32 35 30 62 36 63 66 38 33 33 31 62 37   |682250b6cf8331b7|
0c150   65 65 36 38 66 64 38 52 ea 57 a3 00 00 00 00 80   |ee68fd8R.W......|
0c160   60 5f 5b 57 2f 1e 11 0a 04 00 00 20 13 00 74 00   |‘_[W/...... ..t.|
0c170   00 00 03 00 00 00 01 81 67 cb 00 00 01 73 01 2a   |........g....s.*|
0c180   4b 65 76 69 6e 20 4a 6f 6e 6b 65 72 73 6a 6f 6e   |Kevin Jonkersjon|
```

Original

```
0c090   00 00 00 01 81 67 cb 00 00 01 73 01 10 45 73 61   |.....g....s..Esa|
0c0a0   6e 20 57 69 74 45 73 61 6e 2e 57 69 74 40 6f 73   |n WitEsan.Wit@os|
0c0b0   33 2e 6e 6c 38 62 65 33 63 39 34 33 62 31 36 30   |3.nl8be3c943b160|
0c0c0   39 66 66 66 62 66 63 35 31 61 61 64 36 36 36 64   |9fffbfc51aad666d|
0c0d0   30 61 30 34 61 64 66 38 33 63 39 64 52 ea 57 a3   |0a04adf83c9dR.W.|
0c0e0   00 00 00 00 80 6c 6b 67 63 3b 23 11 0a 04 20 00   |.....lkgc;#... .|
0c0f0   18 13 00 00 00 00 00 02 00 00 00 01 81 76 55 00   |.............vU.|
0c100   00 01 ec 08 55 4c 65 65 6e 64 65 72 74 20 76 61   |....ULeendert va|
0c110   6e 20 44 75 69 6a 6e 4c 65 65 6e 64 65 72 74 2e   |n DuijnLeendert.|
0c120   76 61 6e 44 75 69 6a 6e 40 6f 73 33 2e 6e 6c 35   |vanDuijn@os3.nl5|
0c130   62 61 61 36 31 65 34 63 39 62 39 33 66 33 66 30   |baa61e4c9b93f3f0|
0c140   36 38 32 32 35 30 62 36 63 66 38 33 33 31 62 37   |682250b6cf8331b7|
0c150   65 65 36 38 66 64 38 52 ea 57 a3 00 00 00 00 80   |ee68fd8R.W......|
0c160   60 5f 5b 57 2f 1e 11 0a 04 00 00 20 13 00 74 00   |‘_[W/...... ..t.|
0c170   00 00 03 00 00 00 01 81 67 cb 00 00 01 73 01 2a   |........g....s.*|
0c180   4b 65 76 69 6e 20 4a 6f 6e 6b 65 72 73 6a 6f 6e   |Kevin Jonkersjon|
```

After deletion

Figure 5: Fragment of InnoDB data file in REDUNDANT row format

header	field1field2...fieldN

Figure 6: Generic row format

header_field	integer_field	text_field	text_field	timestamp_field

Figure 7: Example of a possible template

6 Methods

Initial analysis of data files used by the MyISAM and InnoDB engines show that rows are stored in a similar format, see figure 6. Due to this generic form this record may be found using template matching.

6.1 Template Matching

The proposed method uses template matching to extract sequential data from the data files which adhere to a particular template. This template is designed to mirror the on-disk storage structure of a record. An example of a template is given in figure 7. As seen in the figure the template mirrors the generic row format used by the engines in that it specifies a header followed by data fields. If a header is not available this may be omitted in which case the template will merely attempt to match the given fields.

6.1.1 Datatypes

A requirement of template matching is the ability to read and verify binary data from the table. As such the template must specify the expected order of datatypes in a record. The template matcher will parse data and attempt to map the input to the specified datatypes. Due to the differences in storage for datatypes between InnoDB and MyISAM a custom field parser must be made for a each storage engine. This also means that, in our implementation, a template created for InnoDB can not be used for MyISAM.

Since some datatypes have impossible values it is possible to eliminate those. An example of this could be an `ENUM` field within the record. An `ENUM` has all available options specified. As such, any value which is not one of the expected values excludes the existence of an `ENUM` field on that location.

6.1.2 Caveats

One of the caveats of template matching is that it is very dependent on the quality and complexity of the template. The more elaborate and complex the

template is, the fewer possible matches will exist. For instance, attempting to match a template of structure `INTEGER`, `INTEGER` will result in many false positives. But a template describing a structure consisting of `INTEGER`, `ENUM`, `TEXT`, `TIMESTAMP`, `TEXT` will generally have fewer false positives.

Another shortcoming of template matching is the inability to match split records. Database systems do not always store all data sequentially, large records may be split into multiple sections to fit in the available storage. Template matching only works if there is sequential data that matches the template. Although split records are not included in the rest of this research it should be noted that partial templates are allowed and may match fragments of data contained in split records.

6.1.3 Consideration of NULL fields

The storage engine MyISAM uses the record header to determine which fields in a record are NULL, removing them from the data area of a stored record. While this is is of little concern when accessing live data, the deletion of a record overwrites this crucial information as documented in section B.1.1.

To properly carve deleted MyISAM records with NULL fields the set of templates selected for carving should include specific templates which account for all feasible combinations of NULL fields. By not accounting for NULL fields in a record the template matching phase can produce incorrect output and miss records which do not fit due to missing and misaligned fields.

6.1.4 Consideration of record headers

The storage engine InnoDB uses row headers to store the sizes and presence of fields in a record. These headers (see sections C.1.3 and C.1.4) are required to correctly parse variable size types, NULL fields and are used by the database internally for administration.

Since our experiments show (see sections 5.3 and 5.4) that these headers are only partially overwritten we can use them during carving to obtain both the sizes and NULL indicators of each field. The header not only contains a flag to indicate whether a record is deleted but can also be checked for certain patterns not used in any valid record.

6.2 Validation

Template matching may find false positives when matching random or misaligned data. For this reason the use of validators is proposed. A validator is meant to exclude impossible matches and reduce the amount of improbable

matches. Validators may exist for both field validation and row validation. Each validator returns a score between 0 and 1 which is used as a probability value of the record being correct. This allows the validators to be unsure of the validity but still allow the record to be considered valid if other validators agree.

Even validators producing large amount of false positives can be beneficial to use prioritize further, manual, analysis. There is an inherent trade off between the number of false positives and false negatives which should be taken into consideration.

6.2.1 Field validators

A field validator attempt to validate the content of field. Field validation can be based on datatype or content. An example of a datatype validator would be the parsing method itself. As explained in section 6.1.1 some values can not possibly be valid for certain datatypes. Thus the template cannot match on that location and should be ignored. This results in the algorithm moving on to the next possible data and template combination.

Other validators that can be used are content aware validators. These validators describe specifics of the contents of a field. For instance a validator could be defined on a `VARCHAR` field which is used to store email addresses. The validator can then simply check whether or not the content contains valid syntax for an email address. If the content does not then we matched a record which did not have a email address and as such is an incorrect match of the template. Each validator returns a probability score of the field being valid.

A field level validator can be used to limit recovery to only deleted records by having a field validator match on the the correct flag being set in the record header.

6.2.2 Row validators

Finally when taking a step back in granularity you reach the row validators. Each field has successfully been matched to data and passed its individual validators. A row validator can use the fields to validate each other, comparing fields, detecting impossible combinations[4].

This final check is used to produce a score, which will determine if a match is “valid” enough to keep. The score can be based on weighing the probabilities given by field validators, tests which include multiple fields or external tools to validate (parts) of the record.

[4]for example, last login before registration date

If the outcome is over a given threshold then the match is considered valid else the match is logged or discarded. Should a record be accepted by this test the score will be included in the output to aid in further analysis.

6.2.3 Example validators

- Is the last login date after the registration date?
- Is the username considered legal by the system using the database?
- Is an integer within the expected range?
- Is a product price above zero?
- Does the username match the users real name or email address?
- Is this the user that posted message X?
- Is the avatar for a user an image of a known filetype?
- Is an url valid?
- Are the post message and post title related in any way?

6.3 Scanning

In order to carve for the records using template matching our proof of concept uses a scanner which has 2 operational modes, a thorough byte-by-byte sliding window scanner, or a more intelligent mode which skips the data claimed by existing validated records.

The scanner internally uses a rudimentary queue to schedule any location to scan, this queue receives suggestions from the scanner based on its operational mode and whether a record was matched and validated after each attempt.

6.3.1 Consideration of data of interest

Once a record is successfully matched and validated it can be used to speed the scanning process by eliminating locations to check, or even getting an estimate of where the next valid record is stored.

To use a record for this purpose we have determined the following methods:

- Eliminate $Range(Record_{Location} + 1, Record_{Location} + Record_{Size} - 1)$ offsets since they are part of the data of the this record

- Parse and follow the offset/pointer information in a (partially) intact header

- Consider $Record_{Location} + Record_{Size}$

The downside of these optimizations is that when a record is matched incorrectly any correct records might be skipped since they would conflict with the previously found record. In order to get the all possible matches in a file these optimizations should be evaded, or only used to prioritize the scanning engine.

6.4 High level algorithm

Bringing it all together yields the following algorithm:

```
for location ∈ candidates do
  for template ∈ templates do
    record ⇐ parse(location, template)
    if success(record) then
      score ⇐ validate(record)
      if score > threshold then
        results ⇐ results, (location, score, template, record)
        candidates ⇐ candidates, suggestion(record.length)
      end if
    end if
  end for
end for
```

In order to build a practical implementation some constraints should be considered:

- The `candidates` should not accept previously scanned locations to prevent duplicate results and infinite loops

- Carving potentially malicious data sources may require extra consideration to prevent crashes or worse

- The execution may be parallelized in several places, e.g. per location or per template

- When retaining results total memory consumption might become an issue on larger datasets

7 Proof of Concept

Our implementation is written for `Python 2.7`, its system requirements depend heavily on the configuration used and target file to carve.

- Memory, the entire file to be carved is loaded into memory
- Memory, the more matches are encountered the more memory is actively used
- CPU, Faster is generally better
- IO, the program may generate significant amounts of output
- IO, when running in debug mode a large amount of extra print statements will be executed
- The program is a sequential proof of concept and should be expected to behave as such

The code

Our implementation will be hosted on Github under the BSD 3 clause license `https://github.com/esanwit/SQLCarve`.

Usage

For usage and documentation see A.1.

8 Tests

8.1 Small scale, InnoDB Compact with deleted entries

In order to test recovery from an InnoDB table using the compact row format a table was created several test entries, a total of 7 items were inserted and 6 of those were deleted. Running on this small set we expected some false positives however due to the structure containing an enum field we hoped to limit these during testing. For details on the created table see E.2.

For this test the following template was used, full configuration may be found in E.2:

Listing 1 : Template used for InnoDB testing

```
structure_inno = scanner_shared.RowFormat("Inno_Compact_SomeNULL", [
 { "null": False, "varlen": False,
   "name": "header", "type": inno.InnoCompactHeader},
 { "null": False, "varlen": False,
   "name": "id", "type": inno.Int, "signed":True},
 { "null": False, "varlen": False,
   "name": "TransactionID", "type": inno.InnoTransactionID},
 { "null": False, "varlen": False,
   "name": "RollPointer", "type": inno.InnoRollPointer},
 { "null": False, "varlen": True,
   "name": "name", "type": inno.SmallVarchar},
 { "null": True, "varlen": True,
   "name": "last", "type": inno.SmallVarchar},
 { "null": True, "varlen": True,
   "name": "email", "type": inno.SmallVarchar},
 { "null": False, "varlen": False,
   "name": "gender", "type": inno.SmallEnum,
   "enum_map":True, "enum_mapping": {1:"male", 2:"female"}},
 { "null": False, "varlen": False, "name": "other",
   "type": inno.SmallEnum, "enum_map":True,
   "enum_mapping": {1:"male", 2:"female",3:"Wookie",4:"Anon"}},
 { "null": True, "varlen": False, "name": "birthdate",
   "type": inno.Noise, "max_len":4, "min_len":4},
 {"name":"ignoreme", "type": scanner_shared.Null},
 {"null": True, "varlen": False, "name": "favedate",
  "type": inno.Noise, "max_len":4, "min_len":4},
 {"name":"ignoreme", "type": scanner_shared.Null},
])
```

The template shown in listing 1 matches the template as shown in table 1. In order to recover the fields `Birthdate` and `Favedate`, which were of type `timestamp` a `Noise field` (see section A.2.2) was used.

Header	Row Header
Primary key	Signed INT
Internal field	Transaction ID
Internal field	Roll Pointer
Name	Varchar
Lastname	Varchar (Nullable)
Email	Varchar (Nullable)
Gender	Enum
OtherEnum	Enum
Birthdate	Timestamp, not parsed, 4 bytes
Favedate	Timestamp, not parsed, 4 bytes

Table 1: Structure matched by template specified in listing 1

	Present	Deleted	False	Total
Template	1	6	5	12
Validated	1	6	0	7
Expected	1	6	0	7

Table 2: Results of template matching on a small test set.

Due to the two `enum` fields there were early validators which eliminated all but 12 from the matched records. As shown in table 2 the row validation managed to eliminate 5 matches which did not adhere to the constrains. The row validator used checked whether the primary key was within the range of 1 to 10000 and if the username field contained ASCII characters. Manual inspection showed that all created and deleted records were correctly recovered.

8.2 World dataset

In order to expose our system to a larger, more realistic dataset it was tested against the "world" database from MySQL[5]. This dataset was adjusted to create both a MyISAM and an InnoDB table. From these tables 100 records were deleted using the normal SQL `delete` statement. For the structure of the table used and the records deleted see section E.3.

[5] `https://dev.mysql.com/doc/world-setup/en/index.html`

8.2.1 MyISAM

To run the software on the MyISAM version of the table we used the following template, full configuration may be found in section E.3.1.

Listing 2 : Template used for MyISAM version World dataset

```
scanner_settings["row_format"] = [
    scanner_shared.RowFormat("Not␣deleted␣City", [
        { "name":"ignoreme", "type": scanner_shared.Null},
        { "name": "header", "type": inno.Noise,
          "min_len": 1, "max_len":1,
          "validator": validate_isamnotdel, "min_validation": 0.5},
        { "name": "ID", "type": isam.Int, "signed": True,
          "validator": validate_id, "min_validation": 0.5},
        { "name": "Name", "type": isam.CharFixed, "char_length": 35},
        { "name": "CountryCode", "type": isam.CharFixed,
          "char_length": 3},
        { "name": "District", "type": isam.CharFixed,
          "char_length": 20},
        { "name": "Population", "type": isam.Int, "signed":True},
    ]),
    scanner_shared.RowFormat("Deleted␣City", [
        { "name":"ignoreme", "type": scanner_shared.Null},
        { "name": "header", "type": inno.Noise,
          "min_len": 7, "max_len":7, "validator": validate_isamdel,
          "min_validation": 0.5},
        { "name": "ID", "type": scanner_shared.Null},
        { "name": "Name", "type": isam.CharFixed, "char_length": 33},
        { "name": "CountryCode", "type": isam.CharFixed,
          "char_length": 3},
        { "name": "District", "type": isam.CharFixed,
          "char_length": 20},
        { "name": "Population", "type": isam.Int, "signed":True},
    ]),
]
```

This template is designed to recover two types of record, both deleted and non deleted. Distinction is required because in the deleted record the primary key field and first 2 bytes of the character field are overwritten.

In early validation a deleted record is identified by the 0 byte in its record header, intact records are validated by a range check on the primary key. The template so far will generate quite some false positives as any null byte can

be seen as a header, and any 4 byte value as an integer.

In order to reduce false positives the record level validators is composed of several simple validations, in this test for the system to consider a match valid all of these need to match:

- Country code, this field needs to be 3 upper case letters
- Population, this number should be between 1 and a rough billion people
- City name, this field may not contain more than 7 spaces prior to the last non whitespace character

Found	17150
Validated	4079
Validations Failed	0
Tried	273295

Table 3: Results of the world dataset for MyISAM table

Test results can be seen in table 3. The template itself matched 17150 locations. Only 4079 of those matches also validated against the validators. Manual analysis showed all records were recovered, both deleted and intact with the correct templates. This was expected as no further `update` or `insert` statements were executed after deletion.

These results also show that strict validation and template matching can be used to recover data from a MyISAM database even when parts of the data are overwritten due to deletion.

8.2.2 InnoDB

In order to run the test on an InnoDB table in compact row format we adjusted the template to match, see E.3.2 for the full configuration used.

Listing 3 : Template used for InnoDB version World dataset

```
scanner_settings["row_format"] = [
  scanner_shared.RowFormat("Inno_City", [
    { "name": "header", "type": inno.InnoCompactHeader},

    { "name": "ID", "type": inno.Int, "signed": True,
      "validator": validate_id, "min_validation": 0.5},
    { "name": "TransactionID", "type": inno.InnoTransactionID},
    { "name": "RollPointer", "type": inno.InnoRollPointer},
```

```
    { "name": "Name", "type": inno.CharFixed, "char_length": 35},
    { "name": "CountryCode", "type": inno.CharFixed,
      "char_length": 3},
    { "name": "District", "type": inno.CharFixed,
      "char_length": 20},
    { "name": "Population", "type": inno.Int, "signed":True},
  ])
]
```

With the validators set to match

- Primary key, this field needs to be the range 1 to 10000
- Country code, this field needs to be 3 upper case letters
- Population, this number should be between 1 and a rough billion people

Furthermore the fields containing the Name and District are assigned a score based on the ratio of expected characters in the string. In essence where a mundane name scores higher than one with unexpected whitespace or other random characters, e.g. $Score(Holland) > Score(\backslash 0a \backslash 0aHolla)$.

This test shows that using early validation in this template reduces the false positives to a manageable number. It also illustrates that the use of context aware validators can reduce the amount of recovered records even further.

Found	9193
Validated	4142
Validations Failed	0
Tried	507922

Table 4: Results of the world dataset for InnoDB table

As can be seen in table 4 the validation phase brings the number of matches down to 4142. However some false positives were also encountered as can be seen by the fact that more then 4079 entries were validated.

Analysis shows that all our deleted record were recovered and intact, this was expected as only the `delete` statement was executed.

Some false negatives were encountered due to one of the validators not scoring foreign characters correctly.

The false positives returned by the scanner were duplicate, intact records which can possibly be explained by internal reorganization as InnoDB attempts to retain efficiency. During reorganization records are copied to other

pages and the original page is marked as empty although the record remains on the page[10].

9 Conclusion

We set out to recover deleted data from two popular storage engines used by MySQL doing so in a manner which can easily be adapted for other databases. By using template matching we could recover deleted records in the data files used by InnoDB and MyISAM, recovering all data until overwritten by subsequent `insert` or `update` statements. Because InnoDB does not overwrite data belonging to the fields upon deletion it is capable of fully recovering a deleted InnoDB record. MyISAM records however are only partially recoverable due to the deletion header which may overwrite data.

Though template matching shows it is sensitive to generating large numbers of false positives, more so for smaller and simpler templates, validation on both field and row level can bring these down by introducing context aware checks. Simple checks such as ranges for integers and textual context for strings have already shown their effectiveness.

Because the algorithm works directly on the data files it is possible to recover data from tables which have become corrupted and impossible to read by the database itself. Although the corrupted record itself may be missed by the algorithm all intact records may be extracted.

As shown by the proof of concept both supported engines share the same codebase, diverging only in the implementation of raw storage types and available header data. This allows for easy extension to other database systems which use a layout of record data similar to that of MySQL.

10 Further research

Though the proof of concept supports a large number of fields the `BLOB` formats for InnoDB remain unimplemented. This would, combined with support for split records, be a valuable addition to consider.

The generic nature of our solution should allow the extension to other database engines or systems, though many of the storage types are engine specific the recovery method is similar. By separating the implementation from the carving method itself this should allow for easy integration of further fields and structures.

There is a lot of potential for strong and complex validators, with simple validators the number of mismatched entries was reduced by catering to the specific context. If the context is only loosely defined some generic but advanced validators might assign scores based on:

- Text fields containing words, sentences, text
- Username fields obeying typical naming conventions
- Multiple textual fields in a records being the same language
- Numerical fields conforming to statistical properties of known valid records

Also it might be possible to use machine learning to create validators based on data stored in active records. These validators can then be used to validate data thought to belong to deleted records.

One of the most time consuming tasks when using the developed proof of concept is the creation of templates to match the tables. A start was made into using the `FRM` files created by MySQL to automate part of the process. And although the FRM file contains the necessary information there was insufficient time to implement a proof of concept for parsing these files.

References

[1] Patrick Stahlberg, Gerome Miklau, and Brian Neil Levine. Threats to privacy in the forensic analysis of database systems. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 91–102, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-686-8. doi: 10.1145/1247480.1247492. URL `http://doi.acm.org/10.1145/1247480.1247492`.

[2] Ivo Pooters, Pascal Arends, and Steffen Moorrees. Extracting sqlite records - carving, parsing and matching. 2011.

[3] Percona LLC. Percona data recovery tool for innodb. `http://www.percona.com/software/mysql-innodb-data-recovery-tools`.

[4] Martin S. Olivier. On metadata context in database forensics. *Digital Investigation*, 5(3-4):115 – 123, 2009. ISSN 1742-2876. doi: http://dx.doi.org/10.1016/j.diin.2008.10.001. URL `http://www.sciencedirect.com/science/article/pii/S1742287608000972`.

[5] Peter Frühwirt, Peter Kieseberg, Sebastian Schrittwieser, Markus Huber, and Edgar Weippl. Innodb database forensics: Enhanced reconstruction of data manipulation queries from redo logs. *Information Security Technical Report*, 17(4):227 – 238, 2013. ISSN 1363-4127. doi: http://dx.doi.org/10.1016/j.istr.2013.02.003. URL `http://www.sciencedirect.com/science/article/pii/S1363412713000137`. Special Issue: {ARES} 2012 7th International Conference on Availability, Reliability and Security.

[6] Murilo Tito Pereira. Forensic analysis of the firefox 3 internet history and recovery of deleted {SQLite} records. *Digital Investigation*, 5(3-4):93 – 103, 2009. ISSN 1742-2876. doi: http://dx.doi.org/10.1016/j.diin.2009.01.003. URL `http://www.sciencedirect.com/science/article/pii/S1742287609000048`.

[7] P. Kieseberg, S. Schrittwieser, M. Mulazzani, M. Huber, and E. Weippl. Trees cannot lie: Using data structures for forensics purposes. In *Intelligence and Security Informatics Conference (EISIC), 2011 European*, pages 282–285, 2011. doi: 10.1109/EISIC.2011.18.

[8] Simson L. Garfinkel. Carving contiguous and fragmented files with fast object validation. *Digital Investigation*, 4, Supplement(0):2 – 12, 2007. ISSN 1742-2876. doi: http://dx.doi.org/10.1016/j.diin.2007.06.

017. URL http://www.sciencedirect.com/science/article/pii/S1742287607000369.

[9] Oracle. Mysql 5.6 reference manual. https://dev.mysql.com/doc/refman/5.6/en/index.html, .

[10] Oracle. Mysql 5.6 internals manual. http://dev.mysql.com/doc/internals/en/index.html, .

[11] IEEE Computer Society. Ieee standard for floating-point arithmetic. August 2008. doi: 10.1109/IEEESTD.2008.4610935. URL http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.

A Documentation

A.1 Scanner usage and configuration

The scanner tool is executed by running `python2.7` to execute `scanner.py` with a configuration file as an argument.

```
python2.7 scanner.py prepared_config.py
```

The scanner will start by reading the `prepared_config.py` file for all required parameters.

The configuration files used by the Scanner tool are python code, and will most likely use a header with these entries

Listing 4 : Common config header

```
import config_base
import validators
import field_inno_antelope as inno
import field_isam as isam
import scanner_shared
```

The configuration settings are imported from the file as the variable `scanner_settings`. By importing the default configuration only the relevant settings need to be adjusted.

Listing 5 : Scanner_settings

```
scanner_settings = config_base.base_config()

format_a = scanner_shared.RowFormat("Format_a", [
 {"name": "header", "type": inno.InnoCompactHeader},

 {"name": "id", "type": inno.Int, "signed": True},

 {"name": "TransactionID", "type": inno.InnoTransactionID},
 {"name": "RollPointer", "type": inno.InnoRollPointer},

 {"name": "name", "type": inno.SmallVarchar,
  "varlen": True
 },
 {"name": "email", "type": inno.SmallVarchar,
  "null": True, "varlen": True
 }
```

```
])

scanner_settings["filename"] = 'test.ibd'
scanner_settings["row_format"] = [format_a]
scanner_settings["everybytemode"] = True
```

A.1.1 Settings

The system uses several settings to determine its behavior.

Required

filename The filename to carve from

row_format An array of row formats to use as templates

Optional

Debug Whether to print a lot of data during execution

everybytemode Whether to add 1 to the to carve offset after each attempt or use matched record length when available

PrintRecords Whether to print the found records when done

PrintStats Whether to print success rate when done

accept_score The minimal score to fail a validated record

row_validator The function to score an entire record

initial_positions An array of offsets to initialize the processing queue used for matching attempts

skip_positions An array of offsets to blacklist from being used for matching attempts

A.1.2 Validators

A row level validator can be used to validate a complete row and assign it a score.

Listing 6 : Row level validator

```
import validators

def validator(entry):
    todo = {
      "name": validators.validate_ascii,
      "email": validators.validate_email_crude
    }
    score = 1.0
    for k,v in todo.items():
        s = v(entry[k])
        score *= s
    return score

scanner_settings["row_validator"] = validator
scanner_settings["accept_score"]  = lambda x: x > 0.5
```

Row level validators are internally passed the record object, which contains all carved fields and a reference to the row format used.

A field level validator is optionally specified in the row format.

Listing 7 : Field level validator

```
import validators

_validate_ID = validators.make_validator_int_range(1, 10000)
def validate_id(f,n):
    return _validate_ID(f)

row_format = scanner_shared.RowFormat("Something", [
...
   { "name": "ID",
     "type": isam.Int,
     "signed": True,

     "validator": validate_id,
     "min_validation": 0.5
   },
...
])
```

Field level validators are passed the field and the field format entry, which can optionally be used to hold extra configuration. In the example above the

later is ignored and a range check is used from the validators file.

A.2 Generic field types

A.2.1 Null field

The `Null` field type can be used for fields which have no data, these can be found by carving with normal fields which the header indicates as being NULL, or even manually included in a template. When included in a template the scanner will not consume any data from the carved stream, it will generate an entry in the output using the `name`.

Use cases for this field type are

- Indicate a carved record is missing data when a template is in development
- Separate Noise fields when no other field separates them
- Give a row validator a consistent record when combining multiple row formats, when some row formats lack fields required for final validation

This type is found in scanner_shared.

A.2.2 Non specific field

If a field type is unknown or the field data is expected to be tampered with the `Noise` field type can be used.

This field type is designed to allow for matching fields which have either a fixed length or a known range of sizes, without having to actually parse or validate the carved data.

Implementation details require a `Noise` field to be succeeded by a non `Noise` field, this field is used to decide when to conclude the search over all attempted lengths. If there are no more fields a `Null` field can be used, see A.2.1.

In order to get the most out of this field type i.e. use the variable length, the next field should have a strict field level validator. By using a fixed size, $min_len = max_len$, a fixed block of data or fields can be matched or ignored.

B Documentation MyISAM engine

The following section describes some of the internal workings of the MyISAM engine and specifically the storage. It lists the currently supported types and

how they are stored and may be read.

B.1 Headers

As described in section 4.1 the MyISAM engine uses a sequential file for storing data. This file may contain either the fixed or dynamic row format. Depending on the format used recovery may become more difficult.

B.1.1 Dynamic row format

The dynamic row format The MySQL internals documentation[6] lists the available header types. These types are important in determining the templates used for matching. After deletion of a record a 20 byte deleted header is written from the beginning of the record and this overwrites data contained in the first n bytes of the record. n is determined by the original header size for the record and the size of the NULL map which follows. The header looks like:

Header	NULL map	Field1Field2...FieldN

The size of the header is between 3 and 16 bytes. But most commonly the header size will be less then 5 bytes for insert only records and between 4 and 15 bytes for updated records. Since this record is overwritten the contents don't really matter too much for the template matching but it must be determined how many fields can be omitted from the template. The fields spanning the first 3 to 17 bytes of field data will be overwritten on deletion. As such multiple templates must be made to match each of those possibilities. Also due to the fact that the NULL map is overwritten on deletion (at least for the last 40 nullable fields) this means that permutations must be made of the template to match all possible scenarios.

[6] `http://dev.mysql.com/doc/internals/en/layout-record-storage-frame.html`

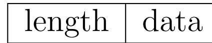

Figure 8: Layout of `blob` storage in MyISAM

B.2 Scanner supported datatypes

SQL type	Scanner type	Configuration required
TINYBLOB	TinyBlob	See B.2.1
BLOB	Blob	See B.2.1
MEDIUMBLOB	MediumBlob	See B.2.1
LONGBLOB	LongBlob	See B.2.1
VARCHAR	VarChar	char_length: ..., see B.2.3
CHAR	CharFixed	See B.2.2
TINYTEXT	TinyText	See B.2.4
TEXT	Text	See B.2.4
MEDIUMTEXT	MediumText	See B.2.4
LONGTEXT	LongText	See B.2.4
FLOAT	Float	See B.2.5
DOUBLE	Double	See B.2.5
REAL	Real	ansi: True or False, see B.2.5
DECIMAL	Decimal	precision: ..., scale: ..., see B.2.6
NUMERIC	Numeric	precision: ..., scale: ..., see B.2.6
TINYINT	TinyInt	unsigned: True or False, see B.2.7
SMALLINT	SmallInt	unsigned: True or False, see B.2.7
MEDIUMINT	MediumInt	unsigned: True or False, see B.2.7
INT	Int	unsigned: True or False, see B.2.7
BIGINT	BigInt	unsigned: True or False, see B.2.7
DATETIME	DateTime	is_packed: True or False, see B.2.8
DATE	Date	See B.2.9
TIME	Time	See B.2.10
TIMESTAMP	Timestamp	See B.2.11
YEAR	Year	See B.2.12
ENUM	EnumField	enum_mapping:["a","b",...], see B.2.13
SET	SetField	set_mapping:["a","b",...], see B.2.14

B.2.1 Blobs

Blobs are stored in MyISAM using the format shown in figure 8. The field header is an integer value describing the length of the following segment. Various sizes are supported for `blob` storage these are shown in table 5

Type	Size
TinyBlob	max 255 (1 byte)
Blob	max 65535 (2 bytes)
MediumBlob	max 16777215 (3 bytes)
LongBlob	max 4294967295 (4 bytes)

Table 5: `blob` column sizes

Type	Size
TinyText	max 255 (1 byte)
Text	max 65535 (2 bytes)
MediumText	max 16777215 (3 bytes)
LongText	max 4294967295 (4 bytes)

Table 6: `text` column sizes

B.2.2 CharFixed

This format requires the `char_length` configuration item, it determines the fixed size of the `char` field in the database.

B.2.3 VarChar

The `varchar` field requires the `varchar_length` option to describe the maximum length this field may hold. The `varchar` is stored in MyISAM the same format as `blob` fields, see figure 8. The field may have either 1 or 2 bytes to store the length depending on the `varchar_length` value.

In the MyISAM fixed row format the storage is padded with null bytes till `varchar_length` is reached.

B.2.4 Texts

This format requires the `charset` configuration item, this determines the character set used for the data stored in the database. All text columns are stored using `blob` storage. The main difference between `blob` and `text` columns is the automatic character set conversion which takes place for `text` columns.

The `text` columns come in 4 sizes mirroring the `blob` layout these sizes can be found in table 6. Due to the character set conversion taking place a `text` field can hold no more than $(2^32) - 1/max_character_length$ characters, where $max_character_length$ is the maximum length any character in the set requires to be stored on disk.

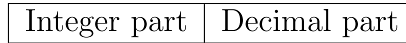

Figure 9: Layout of `decimal` storage in MyISAM

Number of digits	Required size
0	0
1-2	1
3-4	2
5-6	3
7-9	4

Table 7: Number of bytes used to store left over digits.
Source: [9, section 12.19.2]

B.2.5 Float, Double and Real

The `float` and `double` fields are stored using the IEEE 754 standard[11].

The `real` field is an alias for either `float` or `double` depending on options set during compilation of the engine. As such this requires the `ansi` configuration to determine which field type to use.

B.2.6 Decimal and Numeric

The decimal and numeric types are stored exactly the same[9, sections 11.2.2 and 12.19.2]. The parser requires the `scale` and `precision` configuration options to read the storage correctly.

Storage is split into two parts: the integer and decimal part. Both parts are stored sequentially, as seen in figure 9. Each part is a grouping of digits denoting the numerical value contained in the part.

Each group contains at most 9 digits and is stored in a 4 byte integer, contrary to other integers in MyISAM these are stored in Big Endian format. If a part has less then 9 remaining digits the amount of bytes used for storage is also reduced, see table 7 for required sizes.

The left most bit of the integer part is used to denote the sign of the decimal value.

Older versions Versions before 5.0.3 stored the decimal format as a string representation of the value.

Type	Bytes required	Max value
TinyInt	1	255
SmallInt	2	65535
MediumInt	3	16777215
Int	4	4294967295
BigInt	8	18446744073709551615

Table 8: Sizes of `integer` fields in MyISAM

Figure 10: Layout of the compressed `datetime` format

B.2.7 Integers

All integer fields have the `unsigned` configuration options which expresses how the value should be interpreted.

All integers are stored in Little Endian format by the MyISAM format.

The size and max values of each of the types is shown in table 8.

B.2.8 Datetime

The `datetime` field can be stored in two possible formats, a packed and unpacked format.

Packed The packed format stores the data in a 5 byte integer which has the format as shown in figure 10.

The 18 bit compressed value contains the month and year as the modulo and division of the value with 13.

```
month =  value % 13
year = value / 13
```

Unpacked The unpacked format stores the value as its numerical representation So 2014-02-09 15:30:00 would become 20140209153000. This number is then stored in a `BigInt` field

B.2.9 Date

The `date` field is stored in 24 bit. Each part of the date is stored in a section of these 24 bit. The binary layout is shown in figure 11.

Figure 11: Binary layout of a `date` field in MyISAM

B.2.10 Time

The `time` field has two possible formats. Both formats use a 24 bit integer for storage.

Packed The first format is a packed representation. The time of 10 days, 11 hours, 12 minutes and 13 seconds would be stored using the value: 10111213.

Unpacked The second format is an unpacked representation. Here each part of the time field is stored in part of a number. This number is created using the following formula:

$$value = seconds + minutes * 60 + hours * 3600 + days * 24 * 3600$$

The code for reading this value could look something like:

```
value = read(unpacked_time)
days = value / (24 * 3600)
value %= 24 * 3600
hours = value / 3600
value %= 3600
minutes = value / 60
seconds = value % 60
```

B.2.11 Timestamp

A `timestamp` is stored in an unsigned `integer` field as seconds since epoch.

B.2.12 Year

A `year` is stored in an unsigned `tinyint` field as years since 1900.

B.2.13 Enum

The `enum` field stores the selected value as an index of the value list. This index is stored in a `TinyInt` or a `SmallInt` depending on the number of possible values. The list is stored in the table description and only the index is stored in the record.

	Compact	Redundant	Dynamic	Compressed
Antelope	Yes	Yes		
Baracuda	Yes	Yes	Yes	Yes

Figure 12: InnoDB supported row formats per file format

B.2.14 Set

A `set` is stored as a bitmap representation of the possible values. Each possible function is associated with a bit position; the first value is the least significant bit and the last value is the most significant bit. The bitmap reserved is either 1, 2, 3, 4 or 8 bytes long depending on how many items are in the set.

C Documentation InnoDB engine

C.1 Row format

InnoDB supports several row formats, the default being `Compact`.

The Compact and Redundant row formats each have a specific row header, where the Compact header has a variable size based on the presence of certain values in the record.

C.1.1 Primary key

If an InnoDB table does not specify its own primary key field the engine adds a 6 byte row ID field, a unique identifier which is treated as a numerical value.

The number is stored in Big Endian byte order.

C.1.2 Transaction ID and Roll pointer

The InnoDB system uses two special fields in each record.

	Size	Purpose
Transaction ID	6 Bytes	Indicates the latest transaction writing to this row
Roll Pointer	7 Bytes	Points to an undo log record

These fields are modified by a deletion and not normally visible to a user.

C.1.3 Redundant header and Record Directory

The InnoDB Redundant row format uses a record directory to locate fields in a record, it stored as an array of relative pointers[7]. The entry size for these pointers is determined by the size of the entire record where records of $size < 127$ the pointer is a single byte, larger records using a 2 byte Big endian value.

For each possible field an entry in this directory is present, even if those with value NULL, for which the offset is stored $offset|0x80$ for a 1byte entry or $offset|0x8000$ for 2byte entries.

To determine the size of a field I can be determined by calculating $Offset_{I+1} - Offset_I$.

The size of this directory can be calculated:

N := The number of fields for this table
P := $(length(record) < 127)?1 : 2$
Size in bytes := P*N

The record directory is followed by a fixed size header of 6 bytes. This header contains accounting information and flags indicating the state of the record.

C.1.4 Compact header and variable length header

The Compact row format has a variable length header containing a NULL bitmap and the sizes of non-NULL fields in a record. It is stored in front of the fixed size header and entries are stored in reverse order. While a deletion does affect the fixed size header of a row it does not damage the variable size header until the record is overwritten. In order to recover a record with VARCHAR field of NULL fields this requires parsing.

The size of the variable length header is determined by the number of NULLable fields and the number of non-NULL variable length field in the record.

N := Number of nullable fields
M := Number of non-NULL variable length fields with $0 \leq MaxSize < 127$
O := Number of non-NULL variable length fields with $127 \leq MaxSize$
Size in bytes := $Ceiling(\frac{N}{8}) + M + 2 * O$

This variable length header is followed by the fixed size header of 5 bytes, containing accounting information and flags regarding record state.

[7] in reverse order, first field last

C.2 Scanner supported types

The following types are supported for InnoDB Compact and Redundant, to use them include

```
import field_inno_antelope as inno
```

This will result in the objects to become accessible under `inno.` for example `inno.TinyInt`. All fields require a `name` which must be unique within one row format. Duplicate names may crash the program or discard arbitrary fields from the recovered record.

SQL type	Scanner type	Configuration required
Compact Header Redundant Header	InnoCompactHeader InnoRedundantHeader	Required as first field Required as first field, psize: 1 or 2
RowID rollpointer transactionid	Noise InnoRollpointer InnoTransactionID	min_len: 6, max_len: 6
TINYINT SMALLINT MEDIUMINT INT BIGINT	TinyInt SmallInt MediumInt Int Bigint	signed: True or False, see C.2.4 signed: True or False, see C.2.4 signed: True or False, see C.2.4 signed: True or False, see C.2.4 signed: True or False, see C.2.4
ENUM	SmallEnum or BigEnum	enum_map: True or False, enum_mapping:["a","b",...], see C.2.12
CHAR VARCHAR VARCHAR	CharFixed SmallVarchar or BigVarchar VarChar	char_length: ..., see C.2.2 See C.2.2 varchar_length: ..., see C.2.2
DECIMAL NUMERIC	Decimal Numeric	precision: ..., scale: ..., see C.2.13 precision: ..., scale: ..., see C.2.13
DATETIME DATE TIME TIMESTAMP YEAR	DateTime Date Time Timestamp Year	is_packed: True or False, see C.2.14 See C.2.8 See C.2.9 See C.2.10 See C.2.11
FLOAT DOUBLE REAL BIT	Float Double Real Bit	See C.2.7 See C.2.7 ansi: True or False, see C.2.7 length: ..., see C.2.5
SET	Set	set_mapping:["a","b",...], see C.2.6

C.2.1 Nullable fields

If a field is marked as `NULL` in the InnoDB header it may be omitted from the datafile including any length bytes for the Compact row format. Under Redundant the length byte is used to mark the value as being NULL, where Compact uses a dedicated bitmap in the variable length header.

C.2.2 CharFixed

This format requires the `char_length` configuration item, it determines the fixed size of the `char` field in the database.

C.2.3 VarChar

The `varchar` field requires the `varchar_length` option to describe the maximum length this field may hold. The header for a record contains the length of the field, in 1 or 2 bytes depending on the `varchar_length` value.

The `VarChar` type tries to determine the size of this length byte automatically.

C.2.4 Integers

The `tinyint smallint mediumint int bigint` are stored in Big-Endian format, using 1 2 3 4 8 Bytes respectively. This format requires the `signed` configuration item. If the number is of a signed type it will be stored with the first byte Xor 0x80.

C.2.5 Bit

The `bit` field is stores in a bitmap which contains up to 8 entries per byte.

The `length` configuration field determines the number of bit entries.

C.2.6 Set

The `set` field is stored as a `bit` field and for each set bit membership of the set is set.

`set_mapping` configuration field determines the names to display.

C.2.7 Float Double and Real

See B.2.5

C.2.8 Date

The `date` field is stored in a `MediumInt`.

C.2.9 Time

The `time` field is stored in a `MediumInt`.

C.2.10 Timestamp

The `timestamp` field is stored in an `Int`.

C.2.11 Year

The `year` field is stored in an `TinyInt`.

C.2.12 Enum

See B.2.13

C.2.13 Decimal and Numeric

See B.2.6

C.2.14 Datetime

See B.2.8

D Configuration examples

Non validating simple InnoDB record

```
import config_base
import validators
import field_inno_antelope_compact as inno
import scanner_shared

scanner_settings = config_base.base_config()

def validator_a(entry):
    todo = {
      "name": validators.validate_ascii,
      "email": validators.validate_email_crude,
```

```
      "id": validators.validate_int4_x80,
    }
    sco = 0.0
    for k,v in todo.items():
        s = v(entry[k])
        print("Sco<%s> %s"%(k,str(s)))
        sco += s
    return sco

structure_inno_a = scanner_shared.RowFormat("Inno simple", [
 { "name": "header", "type": inno.InnoFixedHeader},
 { "name": "id", "type": inno.InnoInt},
 { "name": "TransactionID", "type": inno.InnoTransactionID},
 { "name": "RollPointer", "type": inno.InnoRollPointer},
 { "varlen": True, "name": "name", "type": inno.InnoSmallVarchar},
 { "null": True, "varlen": True,
   "name": "email", "type": inno.InnoSmallVarchar
 },
 {
   "null": False, "varlen": False,
   "name": "gender", "type": inno.InnoSmallEnum,
   "enum_map":True, "enum_mapping": {1:"male", 2:"female"}
 },
 { "name": "birthdate", "type": inno.InnoInt},
])

scanner_settings["filename"] = 'test.i'
scanner_settings["row_validator"] = validator_a
scanner_settings["accept_score"]  = lambda x: x > 2.5
scanner_settings["row_format"] = [structure_inno_a]
scanner_settings["everybytemode"] = True

scanner_settings["remember_done"] = False
```

The configuration above will look for InnoDB rows with the following structure:

Reading from the file `test.i`, looking at the file byte by byte.

It attempts to validate each possible result using `validator_a` and discard any rows which fail any of the following tests:

- The name field is proper ASCII

- The email address could be valid
- The primary key has 0x80 as its most significant byte[8]

E Data used in tests

E.1 Deletion and header structure

The following tables were used to test what the binary representation does on deletion. After we created the entries in a clean table we made a full copy of the data files, we then deleted one entry from each table and made an other full copy of the data files.

Listing 8 : InnoDB Compact User schema

```
drop table if exists inno_com_user_example;
CREATE TABLE IF NOT EXISTS inno_com_user_example (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE,
    username VARCHAR(20) NOT NULL,
    email VARCHAR(50) NOT NULL,
    password VARCHAR(50) NOT NULL,
    registration_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    last_login TIMESTAMP,
    is_admin BOOLEAN default FALSE
) ENGINE=InnoDB;

INSERT INTO inno_com_user_example(username, email, password) VALUES
  ('Esan Wit', 'Esan.Wit@os3.nl', SHA1('Password')),
  ('Leendert van Duijn', 'Leendert.vanDuijn@os3.nl', SHA1('password')),
  ('Kevin Jonkers', 'jonkers@fox-it.nl', SHA1('1234567'));
```

Listing 9 : InnoDB Redundant User schema

```
drop table if exists inno_red_user_example;
CREATE TABLE IF NOT EXISTS inno_red_user_example (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE,
    username VARCHAR(20) NOT NULL,
    email VARCHAR(50) NOT NULL,
    password VARCHAR(50) NOT NULL,
    registration_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    last_login TIMESTAMP,
    is_admin BOOLEAN default FALSE
```

[8] this property was encountered during early carving attempts

```
) ENGINE=InnoDB ROW_FORMAT=redundant;

INSERT INTO inno_red_user_example(username, email, password) VALUES
   ('Esan Wit', 'Esan.Wit@os3.nl', SHA1('Password')),
   ('Leendert van Duijn', 'Leendert.vanDuijn@os3.nl', SHA1('password')),
   ('Kevin Jonkers', 'jonkers@fox-it.nl', SHA1('1234567'));
```

Listing 10 : MyISAM Fixed User schema

```
drop table if exists isam_stat_user_example;
CREATE TABLE IF NOT EXISTS isam_stat_user_example (
      id INT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE,
      username VARCHAR(20) NOT NULL,
      email VARCHAR(50) NOT NULL,
      password VARCHAR(50) NOT NULL,
      registration_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
      last_login TIMESTAMP,
      is_admin BOOLEAN default FALSE
) ENGINE=MYISAM ROW_FORMAT=FIXED;

INSERT INTO isam_stat_user_example(username, email, password) VALUES
   ('Esan Wit', 'Esan.Wit@os3.nl', SHA1('Password')),
   ('Leendert van Duijn', 'Leendert.vanDuijn@os3.nl', SHA1('password')),
   ('Kevin Jonkers', 'jonkers@fox-it.nl', SHA1('1234567'));
```

Listing 11 : MyISAM Dynamic User schema

```
drop table if exists isam_dyn_user_example;
CREATE TABLE IF NOT EXISTS isam_dyn_user_example (
      id INT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE,
      username VARCHAR(20) NOT NULL,
      email VARCHAR(50) NOT NULL,
      password VARCHAR(50) NOT NULL,
      registration_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
      last_login TIMESTAMP,
      is_admin BOOLEAN default FALSE
) ENGINE=MYISAM;

INSERT INTO isam_dyn_user_example(username, email, password) VALUES
   ('Esan Wit', 'Esan.Wit@os3.nl', SHA1('Password')),
   ('Leendert van Duijn', 'Leendert.vanDuijn@os3.nl', SHA1('password')),
   ('Kevin Jonkers', 'jonkers@fox-it.nl', SHA1('1234567'));
```

E.2 Simple InnoDB compact test

For initial testing on InnoDB compact rows the following table was used:

Listing 12 : InnoDB Compact Row Schema

```
CREATE TABLE IF NOT EXISTS innodb_compact2 (
    id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(100) NOT NULL,
    userlastname VARCHAR(77) ,
    email VARCHAR(100),
    sex ENUM('Male', 'Female') NOT NULL,
    tinder ENUM('Male', 'Female','Wookie','Noneofyourbusiness') NOT NULL,
    birthdate TIMESTAMP,
    favedate TIMESTAMP
) ENGINE=InnoDB;
```

The following configuration was used to detect the simple InnoDB table as described in 8.1.

Listing 13 : Configuration for InnoDB compact test

```
import config_base
import validators
import field_inno_antelope as inno
import scanner_shared

structure_inno = scanner_shared.RowFormat("Inno_Compact_SomeNULL", [
 { "null": False, "varlen": False,
   "name": "header", "type": inno.InnoCompactHeader},
 { "null": False, "varlen": False,
   "name": "id", "type": inno.Int, "signed":True},
 { "null": False, "varlen": False,
   "name": "TransactionID", "type": inno.InnoTransactionID},
 { "null": False, "varlen": False,
   "name": "RollPointer", "type": inno.InnoRollPointer},
 { "null": False, "varlen": True,
   "name": "name", "type": inno.SmallVarchar},
 { "null": True, "varlen": True,
   "name": "last", "type": inno.SmallVarchar},
 { "null": True, "varlen": True,
   "name": "email", "type": inno.SmallVarchar},
 { "null": False, "varlen": False,
   "name": "gender", "type": inno.SmallEnum,
   "enum_map":True, "enum_mapping":
   {1:"male", 2:"female"}},
```

```
 { "null": False, "varlen": False,
   "name": "other", "type": inno.SmallEnum,
   "enum_map":True, "enum_mapping":
   {1:"male", 2:"female",3:"Wookie",4:"Anon"}},
 { "null": True, "varlen": False,
   "name": "birthdate",
   "type": inno.Noise, "max_len":4, "min_len":4},
 {"name":"ignoreme", "type": scanner_shared.Null},
 {"null": True, "varlen": False,
  "name": "favedate",
  "type": inno.Noise, "max_len":4, "min_len":4},
 {"name":"ignoreme", "type": scanner_shared.Null},
])

val_id = validators.make_validator_int_range(1,10000)
val_name = validators.validate_ascii

def validator(entry):
    return val_id(entry["id"]) + val_name(entry["name"])


scanner_settings = config_base.base_config()

scanner_settings["filename"] = 'innodb_compact2.butonedel'
scanner_settings["row_validator"] = validator
scanner_settings["accept_score"]  = lambda x: x > 1.5
scanner_settings["row_format"] = [structure_inno]
scanner_settings["everybytemode"] = True
```

E.3 World dataset

The world dataset table provided by MySQL has the following structure:

Listing 14 : MySQL World dataset

```
CREATE TABLE `City` (
   `ID` int(11) NOT NULL AUTO_INCREMENT,
   `Name` char(35) NOT NULL DEFAULT '',
   `CountryCode` char(3) NOT NULL DEFAULT '',
   `District` char(20) NOT NULL DEFAULT '',
   `Population` int(11) NOT NULL DEFAULT '0',
  PRIMARY KEY (`ID`)
);
```

From this dataset 100 records were deleted, these records were on indices: 2338, 2156, 2996, 1666, 286, 2647, 1298, 2574, 1821, 866, 3400, 3499, 1694, 962, 3905, 176, 867, 543, 2963, 180, 623, 2891, 428, 3466, 293, 2127, 3616, 743, 2353, 1408, 3584, 3748, 1790, 2971, 3874, 1833, 440, 678, 2329, 1963, 1828, 957, 1872, 664, 1118, 3801, 3096, 3117, 1621, 1497, 2343, 2463, 146, 2865, 3341, 1085, 3897, 1788, 790, 2001, 1998, 2633, 1557, 1565, 246, 3146, 600, 2663, 3225, 2396, 944, 2721, 3989, 2668, 3459, 2348, 3237, 3235, 510, 340, 3398, 3619, 476, 3960, 2671, 1117, 1610, 1210, 2167, 2200, 4, 2082, 1950, 1150, 2982, 386, 485, 931, 2456, 1016

E.3.1 MyISAM configuration

The following configuration was used for the MyISAM run on the world dataset.

Listing 15 : Configuration for World dataset MyISAM

```
import config_base
import field_isam as isam
import field_inno_antelope as inno
import scanner_shared

import validators

validate_pop = validators.make_validator_int_range(1,10001000*1000)
def validate_cc(f):
    if validators.validate_null(f):
        return 0.0
    s = f.s

    if all(map(str.isupper, s)):
        return 1.0
    return 0.0

def validate_isamdel(f, n):
    d = map(ord, f.get_raw_data())
    if d[0] == 0:
        return 1.0
    return 0.0

def validate_isamnotdel(f, n):
    return 1.0 - validate_isamdel(f, n)

_validate_ID = validators.make_validator_int_range(1, 10000)
def validate_id(f,n):
    return _validate_ID(f)
```

```
def validate_city(f):
    s = f.s
    if s.rstrip().count('␣') >  7 :
      return 0.0
    return 1.0

def validator(entry):
    score = validate_cc(entry["CountryCode"])
    score += validate_city(entry["Name"])
    score += validate_pop(entry["Population"])
    return score


scanner_settings = config_base.base_config()
scanner_settings["row_format"] = [
scanner_shared.RowFormat("Not␣deleted␣City", [
   { "name":"ignoreme", "type": scanner_shared.Null},
   { "name": "header", "type": inno.Noise,
     "min_len": 1, "max_len":1,
     "validator": validate_isamnotdel, "min_validation": 0.5},

   { "name": "ID", "type": isam.Int, "signed": True,
     "validator": validate_id, "min_validation": 0.5},

   { "name": "Name", "type": isam.CharFixed, "char_length": 35},
   { "name": "CountryCode", "type": isam.CharFixed,
     "char_length": 3},
   { "name": "District", "type": isam.CharFixed,
     "char_length": 20},
   { "name": "Population", "type": isam.Int, "signed":True},
]),
  scanner_shared.RowFormat("Deleted␣City", [
      { "name":"ignoreme", "type": scanner_shared.Null},
      { "name": "header", "type": inno.Noise,
        "min_len": 7, "max_len":7,
        "validator": validate_isamdel, "min_validation": 0.5},

      { "name": "ID", "type": scanner_shared.Null},

      { "name": "Name", "type": isam.CharFixed, "char_length": 33},
      { "name": "CountryCode", "type": isam.CharFixed,
        "char_length": 3},
      { "name": "District", "type": isam.CharFixed,
```

```
        "char_length": 20},
      { "name": "Population", "type": isam.Int, "signed":True},
  ]),

]

scanner_settings["filename"] = 'City_isam.MYD'
scanner_settings["everybytemode"] = True

scanner_settings["row_validator"] = validator
scanner_settings["accept_score"]  = lambda x: x > 2.9
```

E.3.2 InnoDB configuration

The following configuration was used for the InnoDB run on the world dataset.

Listing 16 : Configuration for World dataset InnoDB

```
import config_base
import field_inno_antelope as inno
import scanner_shared

import validators


validate_pop = validators.make_validator_int_range(1, 10001000*1000)
def validate_cc(f):
    if validators.validate_null(f):
        return 0.0
    s = f.s

    if all(map(str.isupper, s)):
        return 1.0
    return 0.0

_validate_ID = validators.make_validator_int_range(1, 10000)
def validate_id(f,n):
    return _validate_ID(f)

def validate_city(f):
    s = f.s
    if s.rstrip().count(' ') > 7 :
      return 0.0
```

```
    return 1.0

def plain_ascii(f):
    s = f.s.rstrip()
    if len(s) < 1:
        return 0.0
    c = len(
        filter(lambda x: x.isalnum() or x in " -,", s)
        )/(1.0*len(s))
    return c

def validator(entry):
    score = validate_cc(entry["CountryCode"])
    score += plain_ascii(entry["Name"])
    score += plain_ascii(entry["District"])
    score += validate_pop(entry["Population"])
    return score


scanner_settings = config_base.base_config()
scanner_settings["row_format"] = [
  scanner_shared.RowFormat("Inno City", [
     { "name": "header", "type": inno.InnoCompactHeader},

     { "name": "ID", "type": inno.Int, "signed": True,
       "validator": validate_id, "min_validation": 0.5},
     { "name": "TransactionID", "type": inno.InnoTransactionID},
     { "name": "RollPointer", "type": inno.InnoRollPointer},

     { "name": "Name", "type": inno.CharFixed, "char_length": 35},
     { "name": "CountryCode", "type": inno.CharFixed,
       "char_length": 3},
     { "name": "District", "type": inno.CharFixed,
       "char_length": 20},
     { "name": "Population", "type": inno.Int, "signed":True},
  ])
]

scanner_settings["filename"] = 'City_inno.ibd'
scanner_settings["everybytemode"] = True
scanner_settings["initial_positions"] = [81904]

scanner_settings["row_validator"] = validator
```

```
scanner_settings["accept_score"]  = lambda x: x > 3.5
```