# USING INTERNAL MySQL/InnoDB B-TREE INDEX NAVIGATION FOR DATA HIDING

Peter Fruhwirt, Peter Kieseberg, Edgar Weippl



Chapter 11

# USING INTERNAL MySQL/InnoDB B-TREE INDEX NAVIGATION FOR DATA HIDING

Peter Fruhwirt, Peter Kieseberg and Edgar Weippl

**Abstract** Large databases provide interesting environments for hiding data. These databases store massive amounts of diverse data, they are riddled with internal mechanisms and data pools for enhancing performance, and they contain complex optimization routines that constantly change portions of the underlying file environments. The databases are valuable targets for attackers who wish to manipulate search results or hide traces of data access or modification. Despite its importance, research on data hiding in databases is relatively sparse. This chapter describes several data hiding techniques in MySQL and demonstrates the impact of data deletion on forensic analysis.

**Keywords:** Databases, data hiding, data deletion, index, InnoDB, MySQL

## 1. Introduction

Strong interest in big data analytics has significantly increased the amount of data that is stored and accessed using high-performance techniques. Databases enhance the performance of operations, especially searching, and enable the reconciliation and linking of large data sets, while supporting the inclusion of many complex operations. Large corporations routinely use data warehouses with workflow engines that automatically enrich raw source data with operational information and data from other data streams. These databases are often massive and provide the foundation for many corporate activities (e.g., financial analysis and billing).

Large databases are perfect places to hide data, more so because they constitute a central and, usually, trusted part of an information technology environment. This trust is usually achieved by applying automated

control systems, sanity checks and audits of the higher software layers, including the input data and results. The database itself usually functions as a black box due to its high level of complexity, massive data content, throughput and relatively opaque internal operations, often introduced to enhance performance. This also results in considerable background noise when examining the deeper layer of file operations; this effectively hinders classical forensic approaches. Furthermore, a database typically holds large amounts of sensitive data, making it easier to hide data inside the database instead of extracting the data and secreting it elsewhere.

Two types of techniques are used to hide data: (i) data removal; and (ii) data disguise. Data hiding seeks to make data inaccessible without leaving any traces while data disguise involves hiding data in other normal-looking data (e.g., using steganography). An important difference exists between data hiding and cryptography. According to Bender et al. [1], the goal of data hiding "is not to restrict or regulate access to the host signal, but rather to ensure that embedded data remain inviolate and recoverable."

The main requirements for data hiding techniques are [1]:

- Access to the hidden or embedded data must be regulated.
- The hidden data must be recoverable.
- The integrity of the hidden data must be ensured.

This chapter proposes several techniques for hiding data in database management systems using index manipulation. A novel approach is presented for evaluating the data hiding techniques. The practical applications of the techniques are showcased using MySQL/InnoDB index mechanisms.

## 2. Background and Related Work

Databases not only store large amounts of information, but also substantial meta-information in order to facilitate fast searches and other operations on tables. Thus, considerable space is allocated that is invisible to database users, but that is, nevertheless, affected by operations and internal mechanisms. This section briefly discusses the use of database meta-information to hide data as well as the related topic of database forensics, especially forensic analyses of the index structure.

Traditional database forensics is mainly focused on analyzing the underlying filesystem layer to recover modified files [6, 13]. Internal mechanisms for guaranteeing database correctness and providing rollback functionality have been used for forensic purposes [4, 5].

Lahdenmaki and Leach [10] provide details of the internal workings of indices at a generic level, as well as related to database management systems and their underlying storage engines. They also discuss the efficient implementation of database indices, which is one of the basic requirements for creating slack space in real-world systems. Further analysis of the internal workings of index trees is provided in [11], where the possibility of using the structures for forensic purposes is also mentioned. Koruga and Baca [9] discuss how B-trees can be used for FAT32 filesystem forensics by searching for remnants of deleted data in the underlying navigation tree. Another approach utilizing the index tree for forensics is discussed in [12]: since the $B^+$-tree for a given set of elements is not unambiguous, the exact structure depends on the order in which the elements are inserted into the tree. Kieseberg et al. [8] have described several scenarios where manipulations of indexed data in a database can be detected by studying the structure of the underlying $B^+$-tree. Based on these observations, Kieseberg et al. [7] have proposed a new logging mechanism.

Pieterse and Olivier [14] have proposed some practical techniques for hiding data in PostgreSQL implementations. The techniques employ the SQL interface to hide structures, which makes them easy to implement, but also easy to detect in forensic investigations. In contrast, this research focuses on techniques that hide data deep inside the internal mechanisms, significantly hindering detection. Furthermore, several layers for manipulating query results are provided, allowing for targeted manipulations (e.g., of automated queries) without changing the results of manual investigations or audit routines.

## 3. InnoDB Index

In InnoDB, data is stored in the form of an index tree based on the primary index, which is mandatory for every table. The data and the primary index are thus closely intertwined and directly affect each other; secondary indices are quite different because they solely exist for the purpose of speeding up specific searches. When creating a table, InnoDB generates an index for the primary key (an auto-incremented id-column is generated if no column is specifically selected). The actual data records are then stored directly inside the $B^+$-tree structure of the index. An additional index tree is generated for each secondary key, this index tree holds pointers to the respective pages in the primary key.

InnoDB uses a $B^+$-tree to locate pages. The first `INDEX` page in the tablespace is called the root node. All the data (keys and data records for the primary key and the corresponding links to the primary key

pages in the case of secondary indices) are stored in the leaf nodes of the tree. All the other pages (i.e., inner nodes of the tree) are only used for navigation and do not contain any user records (note that, in very small tables, the root node may be the (only) leaf node). All the leaf nodes are sorted and are, therefore, implemented as a singly-linked list. For faster navigation within a page, InnoDB uses a page directory that directly links to every fourth to eighth element.

The index is physically stored in pages, which are containers of size of 16 KiB. The pages are stored in tablespace that resides in `ibdataX` files (global tablespace) or in `*.ibd` files if the `file-per-table` feature is active. Each `INDEX` page contains a `FIL` header, which incorporates meta-information about the page itself: an `INDEX` header with a lot of data related to the index, a `FSEG` header with certain pointers, infimum and supremum records and a `FIL` trailer containing checksums.

The user records are located right after the various headers in a page and are physically stored in order of their insertion. Next pointers are used for each record to create an ordered singly-linked list in which the infimum record points to the first record. The user records use the next pointer field to link to the next entry in ascending order. The next pointer of the last user record points to the supremum record that signals the index navigation algorithm that all the records of the page have been read.

## 4. Data Removal

For performance reasons, InnoDB does not physically delete records. In fact, data records persist after deletion as a result of delete flags [2]. These garbage records are overwritten in the future if the space is needed.

As mentioned above, InnoDB uses a singly-linked list for navigation within a page. Specifically, InnoDB uses two `INDEX` header fields: (i) a pointer to the start of the free record list of a page; and (ii) a field that stores the number of bytes of deleted records. Figures 1 and 2 illustrate the deletion process of a data record (note that "@$x$" denotes an $x$-byte page offset, i.e., the physical data address in the filesystem). The garbage offset points to the first deleted record on the current page. As in the case of stored data records, InnoDB uses a singly-linked list for deleted records; the last deleted record points to itself, which signals the end of the list.

### 4.1 Physical Deletion of Data Records

When a record is deleted, InnoDB changes the deleted flag to one. Also, it updates the next pointer of the previous record in ascending

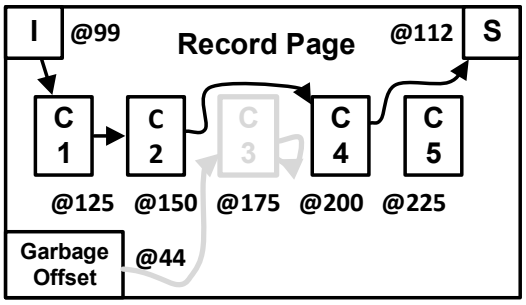


*Figure 1.* One deleted record.

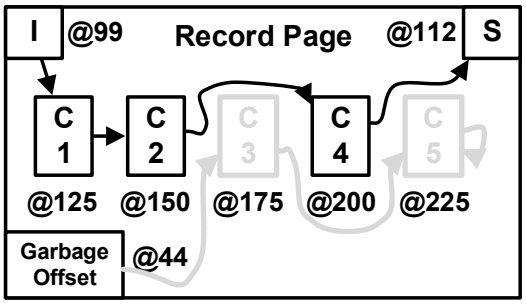


*Figure 2.* Pointer to deleted records in a page.

order and points to the next record or the supremum if the last record on the page is deleted. Additionally, the `INDEX` header is updated, i.e., the garbage size field is increased by the size of the deleted record and the pointer to the last inserted record is overwritten with `0x00000` (Offset: `0x0A`). Internally, the last record of the deleted record list now points to the currently deleted record instead of to itself.

## 4.2 Forensic Impact

Previous research [2–4] has shown that physically-deleted records can be recovered by directly reading the filesystem. Several additional observations can be made based on the index and the actual algorithm that is used:

- **Timeline Analysis:** The design makes it possible to reconstruct the sequence of deletions using the next pointer of the singly-linked free record list of a page. A new deleted record is added to the end of the list.

- **Data Retention:** InnoDB replaces deleted records in the page record free list only if certain conditions are met. First, a new record must be on the same page as the deleted record, which is determined by the structure of the $B^+$-tree. In the case of an auto-incrementing table, this is unlikely because new data records are only on the last page due to the incrementing primary key. If a new record is assigned to the page, InnoDB iterates over the page record free list to find a deleted record with the exact size. If the requirement is not met, InnoDB creates a new page and does not overwrite the deleted records. This method is very efficient, which is important for database management systems. However, it results in long retention times for deleted records, which is excellent from the point of view of digital forensics. Only a complete table recreation or table reorganization force InnoDB to overwrite deleted records.

- **Slack Space:** InnoDB heavily uses pointers for navigation within pages. It is, therefore, possible to manipulate pointers to create areas that are not accessed by the storage engine, thereby creating slack space within stored files. This feature creates the basis for data hiding.

## 5. Data Hiding

This section shows how the structure, and especially the removal mechanism, of the index can be used to create slack space for hiding data. The section also shows how to recover the hidden data. Several techniques for hiding data using the index are described along with their benefits and shortcomings.

### 5.1 Manipulating Search Results

In large-scale databases, data is usually retrieved with the help of secondary (search) indices to provide the desired performance. The dependency on the index can be used to hide data by making it invisible to common searches without actually removing the data from the table. This works by unlinking the index entries that point to the data hidden in the table from the rest of the index, but without modifying the underlying table. If this is done for all relevant searches and their indices, then the data is not retrievable via normal operations. However, the data can be accessed using `SELECT` statements that do not use modified indices or any index at all.

InnoDB uses two types of indices: (i) primary indices, where each table has exactly one primary index that is applied to the primary key

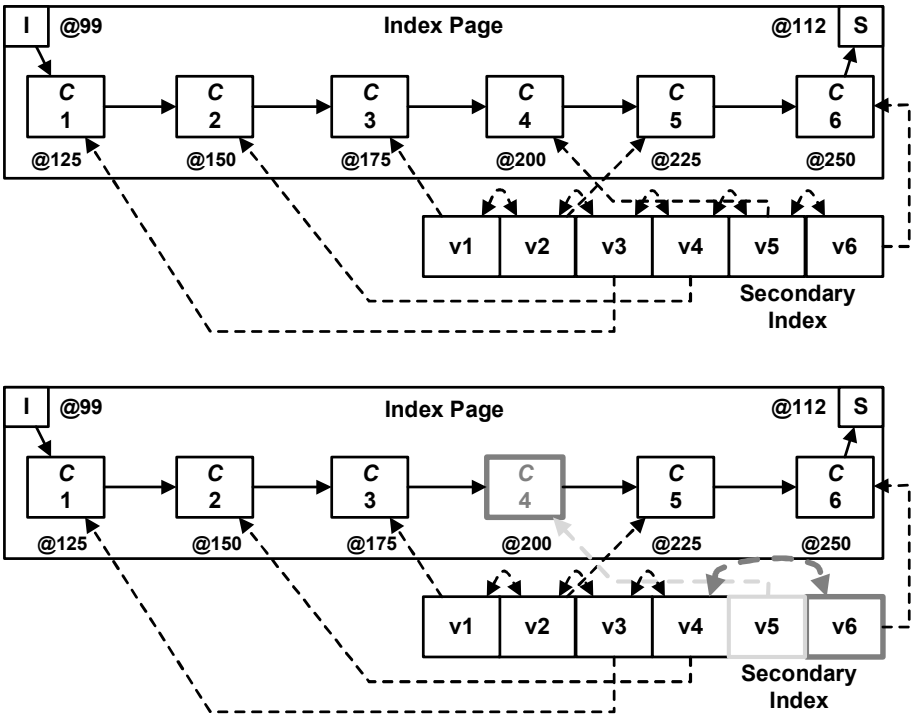


*Figure 3.* Manipulating search results.

of the table; and (ii) secondary indices that are used to enhance search performance. In this approach, the primary index is left unchanged and only the secondary indices are modified. This is reasonable because the primary index is usually an auto-incremented unique field that is not used for actual data retrieval in large databases.

Figure 3 shows how a secondary index can be manipulated to hide data. The index contains a copy of the indexed columns with pointers to the actual record pages where the records are stored. In order to hide a record, the links in the secondary index leading to and from the record and its neighbors are removed and replaced with a direct link between the two neighbors (e.g., $v_5$ in Figure 3 is unlinked from the tree structure and a new link is set to connect its former neighbors, $v_4$ and $v_6$). The record no longer exists in the search tree (secondary index) although it has not been removed from the primary index.

The following general approach is used to hide data:

- The table that will contain the hidden data is generated or selected. The primary index should be chosen in a way that makes

it unsuitable for normal searches (e.g., by adding a generic auto-incrementing id-column that possesses the uniqueness property).

- Secondary indices are generated for all SQL queries used to access table data during normal operations.
- The data to be hidden is written to the table in the form of table entries.
- The links to the data to be hidden are removed from the secondary indices using the approach described above. It is vital that the manipulation of the page index is not omitted.
- The hidden data may be accessed using the primary index or an unmanipulated secondary index.

The main drawback of this approach is that the hidden data can be found by searching via the primary index or by employing unindexed searches. Nevertheless, in many real-world applications (e.g., data warehouses), all the queries involved in the extraction workflows are indexed to obtain the desired performance (e.g., for data cubes).

## 5.2 Reorganizing the Index

While the approach discussed in the previous section has merits, the actual pages holding the table data are still accessible by the database interface; only searching for them using secondary indices is thwarted. The countermeasures are to simply drop and recreate indices regularly or to use searches based on non-indexed columns or the primary index. Therefore, alternative techniques are proposed for hiding data inside the actual index pages. These techniques reorganize the next pointers to create slack space and force the database to skip the hidden records.

Figure 4 shows how a primary index can be manipulated to hide data. The index contains the actual data records, which are linked using the next pointers. In order to hide a record, the links in the primary index leading to/from the record and its neighbors are removed and replaced by a direct link. For example, $C_8$ in Figure 4 is unlinked from the tree structure and a new link is set to connect its former neighbors, $C_7$ and $C_9$. Also, the record must be removed from the page directory, which has a direct pointer to every fourth to eighth element to support faster searches within the page. Note that the directory has to be reorganized if the hidden record is referred to by the page directory.

The following general approach is used to hide data:

- The table that will contain the hidden data is generated or selected.

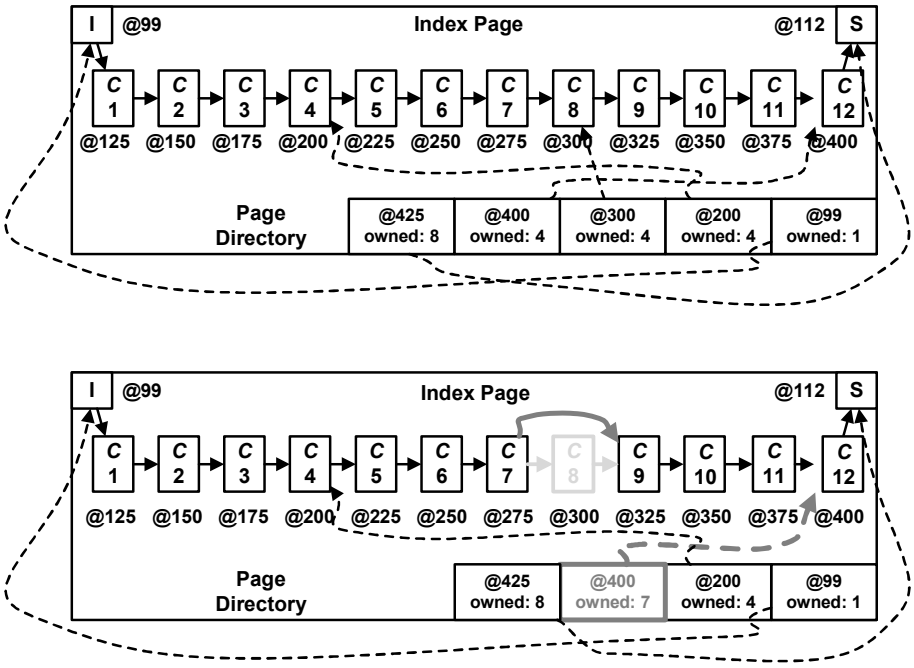


*Figure 4.* Reorganizing the index.

- The data to be hidden is written to the table in the form of table entries.
- The links to the data to be hidden are removed from the primary indices using the approach described in Section 5.1.
- The hidden data may be retrieved using classical forensic methods such as file carving on database records [2] or by reorganizing the pointers and using them to access the hidden data.

Note that the hidden records are still accessible via the secondary indices and have to be further hidden using the technique described in Section 5.1.

## 5.3 Hiding Data in Index Page Garbage Space

This section describes techniques for hiding data within the index pages, thus removing the data from the table altogether. These techniques can be seen as extending the original approach to the primary index. The main difference when removing a primary index compared with a secondary index is that the primary index constitutes the table content (i.e., all records belonging to a table are indexed by the pri-

mary index and any record removed from it is also removed from the table). The principal danger in manipulating the primary index is the generation of inconsistencies that not only enable manipulations to be detected, but also potentially destroy the correctness of large portions of the database.

Section 4 described the internal workings of data removal from the primary index by unlinking the record in the tree and linking it to the list of deleted records starting with the garbage offset. Section 5.1 described how secondary indices can be modified to manipulate searches. Thus, there are two starting points for hiding data from the primary index. First, it is possible to manipulate the delete operation in order to not link the deleted record to the list of deleted records; thus, the space containing the hidden data is not overwritten by the database. Second, the approach of unlinking the record in the secondary indices can be extended to the primary index; this links the neighboring records and removes the links to the hidden record. In the case of the second approach, it is also necessary to remove the links in all the secondary indices so as not to create inconsistencies in the database.

This approach can be extended to the primary index data hiding mechanisms (Section 5.2). Specifically, the hidden record stored in the table is unlinked as in the case of deletion, but it is not linked to the garbage collection. This also involves unlinking the record in every secondary index.

The following generic approach can be used to remove the data to be hidden from the primary index and, thus, from the table:

- The table that will contain the hidden data is generated or selected. No requirements are imposed on the primary index, especially related to its use in searches.
- The record holding the data to be hidden is removed from the table using a modified version of the delete operation (see the next step).
- When deleting the associated entry in the index tree, a modification of the delete operation is applied. While the record is unlinked from the tree as is done normally, it is not linked to the list of deleted records; thus, it is not marked as being available for future use. Since this is the only change with respect to the original deletion mechanism, all the secondary indices are updated normally.
- The hidden data may be retrieved using classical forensic methods such as file carving.

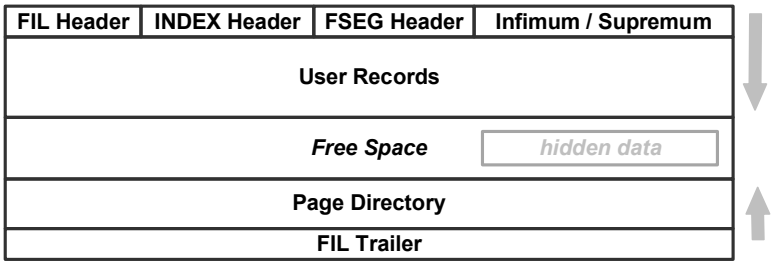


*Figure 5.* Physical structure of an index page.

The drawback of this method is that the insert and delete operations leave traces in the transaction log and other locations. This drawback is addressed by not hiding the data in the underlying table, but, instead, storing some arbitrary, unsuspicious data. Then, after unlinking the record from the index, the free space is filled with the data to be hidden using file carving. This has the additional benefit that the data can be changed later (i.e., the method is only used to create slack space that is not allocable by the database).

Unfortunately, navigation by file carving is rather tedious and inefficient if many reads need to be done. To enhance the usability of the slack space, a further enhancement is necessary. Analogous to the linked list of free space that starts with the garbage offset, the enhancement involves linking all the generated slack space in a page starting with a hidden page offset and a link to the first hidden record. The last record links to itself in order to signal the end of the list. This allows for easy navigation through the slack space in a page, because only the hidden page offset needs to be found by file carving.

Hidden page offsets can be linked together to further enhance searches in the slack space. This is done by generating a $B^+$-tree and creating a shadow index much like the primary index of a regular table.

## 5.4 Hiding Data in Index Page Free Space

Due to the physical structure of a page, some free space exists that is allocated by the storage engine but not used (see Section 3). Figure 5 shows the structure of an index page. New records are written to the user record space towards the `FIL` trailer in the order of their insertion. Simultaneously, the page directory grows towards the user records. If the two sections meet, the free space of the page is exhausted and the page is considered to be full. This free space is not used by the database management system and can be used to hide data. However, unlike the other data hiding techniques, this technique does not protect

against overwriting because the database management system considers the space to be free.

## 5.5 Removing a Page from the Index

InnoDB uses pointers between pages to create a $B^+$-tree. These pointers are used to find the page where the requested data is stored. All the leaf nodes contain the actual record data, unlike the non-leaf nodes that only contain pointers to the next pages. All the pages at the same level are doubly-linked to their predecessors and successors. As in the case of index reorganization (Section 5.2), it is possible to change the pointers to unlink a page and use it to hide data. However, our experiments revealed that this approach is infeasible because, in general, a regular page contains considerable data that is also referred to by the secondary indices, which results in many additional updates. Furthermore, the $B^+$-tree has to be rearranged, which creates massive overhead.

According to the internal source documentation of InnoDB, the database storage engine accesses every data record exclusively via the primary index. If a record is not accessible via this index, the data record does not exist as far as the database management system is concerned. This architecture makes sense with regard to performance, but it can be misused for data hiding purposes as described above.

Since it would be imprudent to rely solely on the source code documentation, a new method was created in order to evaluate the data hiding techniques. The basic idea is to create a set of queries that are executed on a manipulated table space. A check is made if a test token that was hidden earlier can be retrieved and if the storage engine crashes as a result of the data hiding technique. It is impossible to cover all possible query combinations. However, it is feasible to use a fuzzy testing approach that generates a large test set that simulates a real-world environment and, because the navigation algorithms are limited, it is possible to guarantee with some certainty that all the query combinations are covered.

The `randgen` SQL generator (`launchpad.net/randgen`) was used in the evaluation. Originally designed for functional and stress testing, this tool implements a pseudorandom data and query generator. Test cases were generated using a context-sensitive grammar and input to `randgen`.

In the evaluations, only queries that actually rely on the existence of data (i.e., that reveal the existence of the hidden token) were considered. The following operations fulfill this requirement:

- `SELECT` operations
- `JOIN` operations

- `INSERT` operations with `ON DUPLICATE UPDATE`

Note that no functions, procedures, triggers, sub-queries or views were used. This is because they are handled by the same internal functions and mechanisms as the operations listed above.

The following (simplified) context-sensitive grammar specifying the `SELECT` syntax was used:

```
SELECT
  [ALL | DISTINCT | DISTINCTROW ] [SQL_CACHE | SQL_NO_CACHE]
  select_expr [, select_expr ...]
  [FROM table_references
  [WHERE where_condition]
  [GROUP BY {col_name | expr | position}
    [ASC | DESC], ... [WITH ROLLUP]]
  [HAVING where_condition]
  [ORDER BY {col_name | expr | position}
    [ASC | DESC], ...]
  [LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

To generate a valid and adequate test set, the `where_condition` was used to force the database management system and the query optimizer to (preferably) use different types of index navigation (i.e., SQL caches, direct access and area searches).

The evaluation procedure executed concrete SQL statements derived using the grammar on a manipulated table containing the hidden token. A test case failed if the hidden token was not retrieved. When a failure occurred, MySQL internal tools such as `EXPLAIN` queries were used to determine the navigation algorithms that were used and the results were grouped into different categories: full-table scans (e.g., `SELECT` statements without an index or `SELECT` statements without `WHERE` conditions); direct access using a primary key (i.e., `JOIN` operations and `WHERE` conditions using primary key fields); and indirect access using secondary indices (i.e., `WHERE` conditions using a secondary index); and area scans using conditions (e.g., `BETWEEN`). Table 1 presents the evaluation results, including the availability of the hidden token using different queries and if the data hiding is persistent and resistant to accidental overwriting by the database management system.

Note that classical SQL uses full-table scans to create data backups. Due to its design, all the index information is lost; therefore, some modifications such as the manipulation of the search results would not be uncovered in digital forensic investigations. Such data tampering can only be detected by examining the live system. However, this is rarely done because of possible side effects to the production system.

*Table 1.* Hidden data accessible via an SQL interface.

| | Full Table | Primary | Secondary | Persistent |
|---|---|---|---|---|
| Manipulating the search results | Yes | Yes | No | Yes |
| Reorganizing the index | No | No | Yes | Yes |
| Hiding data in index garbage space | No | No | No | Yes |
| Hiding data in index page free space | No | No | No | No |
| Removing a page from the index | Yes | Yes | No | Yes |

## 6. Conclusions

This research has demonstrated how indices in InnoDB can be manipulated in order to hide data. Five data hiding techniques were proposed, each with different characteristics and benefits. The techniques manipulate the underlying index structures, making it possible to hide data as well as create free slack space in InnoDB. Depending on the technique, the data may retrieved via an SQL interface by issuing suitable `SELECT` statements or by using advanced file carving methods.

An important practical application is the ability to adjust secondary indices so that hidden data still resides in database tables, making it available for sanity checks and forensic investigations. This is especially useful in the case of large data warehouses used for automated workflows. While the statements used in workflow routines are typically indexed to guarantee the desired performance, manual investigations usually target unindexed searches. Thus, it is possible to manipulate a database so that the hidden data is not accessible by indexed searches and, thus, also by the actual workflow, while making the manipulations invisible to sanity checks and forensic investigations. This also demonstrates that the results returned from a database using an SQL interface cannot be trusted.

The research also shows that an arbitrary amount of hidden slack space can be created in a database. The slack space cannot be searched or modified via an SQL interface and is, therefore, stable with respect to normal database operations. The slack space is especially valuable because it resides directly inside normal data files that are continually changed during normal operations, making additional changes practically impossible to detect via traditional digital forensic techniques (although file carving can be used to access the hidden data [2]). Additional

structures are also implementable in this slack space in order to boost performance.

In conclusion, it is possible to manipulate and hide data inside databases with a potentially large impact on operations in data warehouses as well as on traditional filesystem-based forensics. Future research will extend the techniques to other prominent database management systems and will conduct large-scale case studies involving corporate databases.

## Acknowledgement

This research was supported by the Austrian Research Promotion Agency (FFG) under the Austrian COMET Program and the Hochschuljubiläumsstiftung der Stadt Wien.

## References

[1] W. Bender, D. Gruhl, N. Morimoto and A. Liu, Techniques for data hiding, *IBM Systems Journal*, vol. 35(3-4), pp. 313–336, 1996.

[2] P. Fruhwirt, M. Huber, M. Mulazzani and E. Weippl, InnoDB database forensics, *Proceedings of the Twenty-Fourth IEEE International Conference on Advanced Information Networking and Applications*, pp. 1028–1036, 2010.

[3] P. Fruhwirt, P. Kieseberg, K. Krombholz and E. Weippl, Towards a forensic-aware database solution: Using a secured database replication protocol and transaction management for digital investigations, *Digital Investigation*, vol. 11(4), pp. 336–348, 2014.

[4] P. Fruhwirt, P. Kieseberg, S. Schrittwieser, M. Huber and E. Weippl, InnoDB database forensics: Reconstructing data manipulation queries from redo logs, *Proceedings of the Seventh International Conference on Availability, Reliability and Security*, pp. 625–633, 2012.

[5] P. Fruhwirt, P. Kieseberg, S. Schrittwieser, M. Huber and E. Weippl, InnoDB database forensics: Enhanced reconstruction of data manipulation queries from redo logs, *Information Security Technical Report*, vol. 17(4), pp. 227–238, 2013.

[6] A. Grebhahn, M. Schaler and V. Koppen, Secure deletion: Towards tailor-made privacy in database systems, *Proceedings of the Fifteenth Conference on Database Systems for Business, Technology and Web*, pp. 99–113, 2013.

[7] P. Kieseberg, S. Schrittwieser, L. Morgan, M. Mulazzani, M. Huber and E. Weippl, Using the structure of $B^+$-trees for enhancing logging mechanisms of databases, *International Journal of Web Information Systems*, vol. 9(1), pp. 53–68, 2013.

[8] P. Kieseberg, S. Schrittwieser, M. Mulazzani, M. Huber and E. Weippl, Trees cannot lie: Using data structures for forensic purposes, *Proceedings of the European Intelligence and Security Informatics Conference*, pp. 282–285, 2011.

[9] P. Koruga and M. Baca, Analysis of B-tree data structure and its usage in computer forensics, *Proceedings of the Central European Conference on Information and Intelligent Systems*, 2010.

[10] T. Lahdenmaki and M. Leach, *Relational Database Index Design and the Optimizers*, John Wiley and Sons, Hoboken, New Jersey, 2005.

[11] H. Lu, Y. Ng and Z. Tian, T-tree or B-tree: Main memory database index structure revisited, *Proceedings of the Eleventh Australasian Database Conference*, pp. 65–73, 2000.

[12] G. Miklau, B. Levine and P. Stahlberg, Securing history: Privacy and accountability in database systems, *Proceedings of the Third Biennial Conference on Innovative Data Systems Research*, pp. 387–396, 2007.

[13] P. Stahlberg, G. Miklau and B. Levine, Threats to privacy in the forensic analysis of database systems, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 91–102, 2007.

[14] H. Pieterse and M. Olivier, Data hiding techniques for database environments, in *Advances in Digital Forensics VIII*, G. Peterson and S. Shenoi (Eds.), Springer, Heidelberg, Germany, pp. 289–301, 2012.