Trees Cannot Lie: Using Data Structures for Forensics Purposes

Peter Kieseberg, Sebastian Schrittwieser, Martin Mulazzani, Markus Huber and Edgar Weippl
SBA-Research - Vienna, Austria
Email: sschrittwieser,pkieseberg,mmulazzani,mhuber,eweippl@sba-research.org

***Abstract*—Today's forensic techniques for databases are primarily focused on logging mechanisms and artifacts accessible in the database management systems (DBMSs). While log files, plan caches, cache clock hands, etc. can reveal past transactions, a malicious administrator's modifications might be much more difficult to detect, because he can cover his tracks by also manipulating the log files and flushing transient artifacts such as caches. The internal structure of the data storage inside databases, however, has not yet received much attention from the digital forensic research community. In this paper, we want to show that the diversity of $\mathcal{B}^+$-Trees, a widely used data structure in today's database storage engines, enables a deep insight of the database's history. Hidden manipulations such as predated INSERT operations in a logging database can be revealed by our approach. We introduce novel forensic techniques for $\mathcal{B}^+$-Trees that are based on characteristics of the tree structure and show how database management systems would have to be modified to even better support tree forensic techniques.**

***Index Terms*—database forensics, b+ tree, InnoDB**

I. Introduction

Modern database systems have a very high traceability of modifications because of intense logging mechanisms. Database storage engines such as InnoDB for MySQL store every single manipulation statement in their log files, thus making hidden manipulation of database records a difficult task. However, if tracks of manipulations are removed from log files, traditional database forensic techniques are ineffective, because they are limited to mainly data file and log file analysis. A malicious database administrator, who has full access to the database and the log files, can effectively cover tracks of manipulations.

In this paper, we introduce a novel forensic technique on the $\mathcal{B}^+$-Tree level of the database storage engine based on the ideas described in [1]. The concept of our approach is a sanity check for database content by comparing a stated insertion sequence to the data stored in the tablespace, the transaction logs and the $\mathcal{B}^+$-Tree structure of the indexes.

While a forensic analysis of a $\mathcal{B}^+$-Tree is not likely to contain precise information on which data was modified at what exact time, it can be used very effectively to prove the violation of security and compliance policies. Compliance in a database is usually ensured by access controls, stored procedures, triggers, and audit logs. A database user might only have permissions to insert into a table and not delete. In addition, the user might not have permissions to insert directly into the table but only to execute a stored procedure. This procedure would create additional audit logs, ensure that only increasing time stamps are used and enforce other integrity constraints.

If the current $\mathcal{B}^+$-Tree cannot be derived from the insertion sequence record in the log files or does not match the insertion policy (e.g. strict ordered insert only), a manipulation of the database is very likely. $\mathcal{B}^+$-Tree forensic cannot replace traditional database forensic methods, because evidential quality varies greatly on a case-by-case basis. However, our approach can substantiate evidence collected by other methods.

A second area for its application could be file system forensics. Many of today's files systems such as NFTS, ReiserFS, and BtrFS use $\mathcal{B}^+$-Trees for their internal organization of stored files. Previous forensic methods on file system layer [2], [3], [4], [5], [6] do not analyze the structure of the underlying $\mathcal{B}^+$-Tree, still Koruga et al. [7] reconstructed the $\mathcal{B}$-Trees of filesystems to recover deleted files.

Our main contributions are to show that the structure of a database's $\mathcal{B}^+$-Tree can be a very important characteristic of the insert order of the table and to show that the analysis of a $\mathcal{B}^+$-Tree allows to draw forensic conclusions to disprove an input order extracted from log files.

II. Background and Related Work

A. Traditional database forensics

Today's database forensic is mainly based on data file and log file consistency checking (e.g.[8]). A work by Frühwirt et al. [9] describes the layout of InnoDB's storage files. Log files in InnoDB can recover every single data manipulation statement submitted to a table, because the entire statement is stored. So, an investigator can reconstruct not only older versions of the database, but also manipulation statements that were applied to the database. The data files of InnoDB can be also used for digital forensics. Deleted records are not removed from data files when a user deletes them from the database management system. Instead, they are only flagged as deleted and can be recovered as long as the database storage engine has not overwritten the record with new data.

The InnoDB storage engine makes extensive use of checksums, thus data manipulations are difficult to perform. However, a malicious administrator has full access to log files and therefore they can not be considered reliable.

B. $\mathcal{B}$-Trees and $\mathcal{B}^+$-Trees

A $\mathcal{B}^+$-Tree, according to Bayer [10], is a balanced tree with the following properties:

- Every non-root node contains between $\frac{b}{2}$ and b elements.
- The root node contains b elements at most.
- An inner node with x elements has got $x+1$ child nodes.
- All leaf nodes lie on the same level.
- All elements inside a leaf are sorted.

When an element is added to the $\mathcal{B}$-Tree, the tree is searched for the leaf the element should be placed in. In case this leaf node contains less than b elements, the new element is simply added. If the leaf node contains b elements, the leaf is split into two leafs and the middle element is inserted as a new key into the parent node. In case the parent node contains $b+1$ keys after this operation, the parent node is split too. This is done iteratively until either a parent node contains less than $b+1$ elements, or a new root is formed. In case $b/2$ is an even number, the element with number $b/2+1$ is propagated to the parent node, thus both child nodes will contain $\frac{b}{2}$ elements.

A $\mathcal{B}^+$-Tree of order b is defined like a $\mathcal{B}$-Tree of order b, but all keys reside in the leaf nodes, i.e. all elements in inner nodes are only for referencing purpose and the actual data is stored in the leaf nodes. It has to be defined, if the parent node contains the topping elements of the lower child nodes, or the bottoming elements of the higher child nodes, i.e. it has to be defined in case of splitting a node, whether a copy of the highest element of the child node containing the lower elements is propagated to become a key in the parent node, or the lowest element of the node containing the higher elements.

C. Application in Databases

In databases fast access to data records is crucial. As I/O operations are typically the slowest task in today's computer systems, decreasing required I/O is the primary aim. Modern database storage engines maintain a so-called index that allows fast lookup of records with a minimal amount of I/O operations. In database storage engines such as InnoDB the index builds up a $\mathcal{B}^+$-Trees. The index consists of one or more columns of the table that should have high cardinality for generating a highly structured tree. Compared to other implementations of trees such as $\mathcal{B}$-Trees, $\mathcal{B}^+$-Trees have performance advantages by reducing expensive I/O operations [11]. A high branching level (i.e. a high number of child nodes) reduces the height of the tree and therefore the expensive read operations on nodes.

III. Database Forensics using $\mathcal{B}^+$-Trees

A. Notation and general assumptions

The following notations and general assumptions are used throughout this paper:

- b denotes the maximum number of keys in a node of a given $\mathcal{B}^+$-Tree. We call b the *order* of the tree.
- All nodes of a $\mathcal{B}^+$-Tree except for the root node have a keys with $\lfloor \frac{b}{2} \rfloor \leq a \leq b$ and (in case they are not leaf nodes) $a+1$ child nodes.
- We always propagate the highest element of the lower child to the parent node when splitting a node, i.e. when splitting the root tree $(e_1, \ldots, e_b)$ because of adding the element e_{b+1}, a copy of the element with number $\lfloor \frac{b}{2} \rfloor + 1$ is propagated to become the new root node, thus resulting in the two child nodes $(e_1, \ldots, e_{\lfloor \frac{b}{2} \rfloor + 1})$ and $(e_{\lfloor \frac{b}{2} \rfloor + 2}, \ldots, e_b)$.

B. Forensics on revision secure tables

The need for regulatory compliance (e.g. the US Sarbanes-Oxley Act as well as the European Data Directive on Privacy force companies to effectively protect the access to sensitive business data and enable traceability of business processes) drives the demand for databases that conform to strict limitations on the kinds of operations that are allowed. So, many companies limit access to certain databases to a strict insert-only policy, thus prohibiting deletion and updates on a technical basis. This scenario is the starting point for the forensic approach specified in this section.

The following prerequisites were taken into account:

- Only *INSERT*, but no *UPDATE* and *DELETE* statements are allowed.
- The considered table has a primary key that is constantly incrementing (e.g. timestamp in milliseconds). This key is also used for structuring the $\mathcal{B}^+$-Tree as it creates the index inside the non-leaf nodes.

The following theorem gives us a statement on the structure of the emerging trees:

Theorem III.1. *Let B be a $\mathcal{B}^+$-Tree with $n > b$ elements which are added in ascending order. Then it holds true that the partition of the leafs of B has the following structure:*

$$n = \sum_{i=1}^{k} a_i, \text{ with } a_i = \frac{b}{2} + 1, \forall i \neq k \text{ and } a_k \geq \frac{b}{2}.$$

Proof: We start by inserting b elements into an empty root, thus when inserting the next element we have to split the root and generate a new one with two leafs. The only possibility is to propagate the middle element to the root and split the leafs in one containing $\frac{b}{2}+1$, the other containing $\frac{b}{2}$ elements (else one of the prerequisites for $\mathcal{B}^+$-Trees would be violated. Iteratively when we add elements, it is always added to the rightmost leaf, thus resulting in three cases:

1) The leaf contains less than b elements $\Rightarrow$ the new element is added to the rightmost leaf.
2) The leaf contains b elements. The leaf is split into two leafs, the first (according to the assumption made above that we always propagate the highest element of the lower leafs into the parent node) containing $\frac{b}{2}+1$, the second $b/2$ elements. This can result in two cases:
    a) The parent node is full $\Rightarrow$ it is split itself into two parent nodes containing $b/2$ elements and $b/2+1$ child-nodes each and propagating the middle element into the next level. Eventually this could result in the generation of a new root containing only one single element. Still, no element will be put into leafs that are left of the current leaf.

b) The parent node has enough space, in this case nothing else happens.

Thus, it is impossible for the tree to add further elements into leafs except the rightmost. Again, from this the proposed partitioning follows quite easily. ■

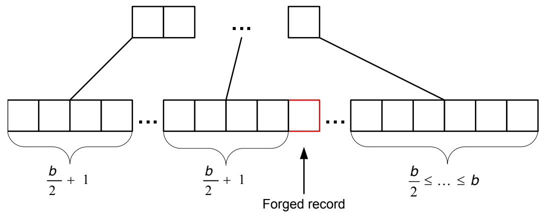

Fig. 1. A $\mathcal{B}^+$-Tree resulting from ascending ordered inserts only.

The main argument for our forensic approach lies in the fact that even with full insert rights, normally, in a database application, a database administrator is not able to change the data at $\mathcal{B}^+$-Tree-level, since this is solely managed by the DBMS itself. Thus at this level, a malicious administrator will not be able to fake evidence.

We will assume that the database does not rearrange the tree elements (e.g. for reasons of performance) and we are able to directly read the structure of the $\mathcal{B}$-Tree (e.g. in InnoDB).

In a $\mathcal{B}^+$-Tree that was built by sorted inserts, non-sorted insert can be detected by analyzing the fill rate of the leaf nodes. If data is inserted in a strictly sorted order (e.g. a logging table with a timestamp as primary key), the fill rate of all leafs except the rightmost one is, according to theorem III.1 exactly $\lfloor \frac{b}{2} \rfloor + 1$ elements.

In case a malicious administrator inserts an additional record with a forged, namely pre-dated timestamp, this record may be added to a leaf node that is not the rightmost one. This happens, if there are already at least $b + 1$ elements with a higher keys (i.e. timestamps) in the table (in case there are less than $b+1$ higher keys in the table, the forged record would be added to the rightmost leaf. On an eventual split, this record would then reside in the correct leaf and additionally this leaf would only contain $\lfloor \frac{b}{2} \rfloor + 1$ elements).

When analyzing the fill rate of leaf nodes in this tree, the pre-dated record is located in a node that has a too high fill rate for strictly sorted inserts (i.e. $> \lfloor \frac{b}{2} \rfloor + 1$ elements). This hidden modification can be detected, because the resulting $\mathcal{B}^+$-Tree does not correspond to the insert policy of the database. Since the elements inside a leaf node are sorted by default, only the leaf containing the forged record can be detected, the identification of the element itself is not possible.

Be aware that not all inserts of this form can be detected by this approach, since with a combination of different forged records, the structure can be fixed again by adding enough (i.e. $\frac{b}{2} + 1$) records (see Figure 2) Actually, in real life tree-sizes, this would result in the insertion of many forged records, which should be detectable by other means (comparisons, sanity checks). Furthermore, these insertions also affect at least the parent node, since a new leaf is generated.

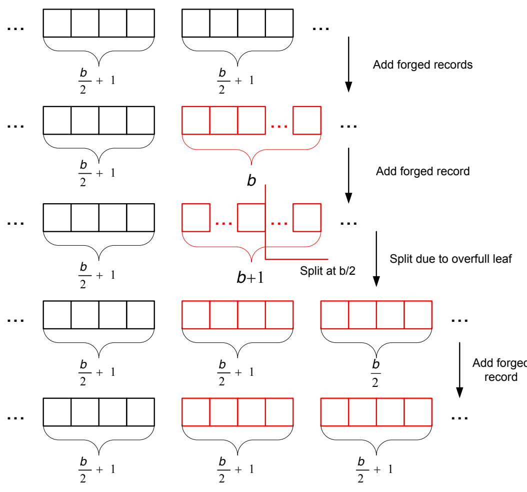

Fig. 2. Forging many records to thwart forensic analysis.

With this approach, Deleted records can be detected in a very similar way. When a malicious administrator deletes one record from a leaf that is not the rightmost, the fill rate of this leaf node falls to $\lfloor \frac{b}{2} \rfloor$ elements, which again is detectable. In case more than one element is removed from the same leaf, the number of elements falls below $\lfloor \frac{b}{2} \rfloor$, thus violating the lower $\mathcal{B}^+$-Tree-boundary for the leaf-size. The following re-balancing will again result in a $\mathcal{B}^+$-Tree-structure, that identifies the manipulation (the merged leafs will result in a leaf of size $> \frac{b}{2} + 1$).

C. Limitations of the approach

The most interesting idea regarding this approach towards database forensics would lie in its generalization to be suitable for all kinds of operations, no matter what order they are applied. Unfortunately we can prove that this is impossible since the $\mathcal{B}^+$-Tree does not give us enough information for this in general. More precisely, the main limitation of our approach lies in the fact that in general the operation of inserting an element into a $\mathcal{B}^+$-Tree is not bijective since the inverse operation is not injective, i.e. even with knowledge on the inserted elements, it can be impossible to recalculate the original tree (or an intermediate state), even though the resulting tree and a log of all operations is available.

Example The example given in Figure 3 illustrates how adding the same element to two $\mathcal{B}^+$-Trees A and A' with different structures generates the same tree B.

Thus the order of insertion (like specified in the revision secure case) is very important for our approach.

D. Towards a forensic-aware database

This section evaluates how today's database management systems have to be modified to provide better forensic evidence. For performance reasons, database management systems tend to generate wide trees with a small number of levels. Our evaluation showed that even with 450,000 records in an InnoDB table, the tree's height did not exceed a value

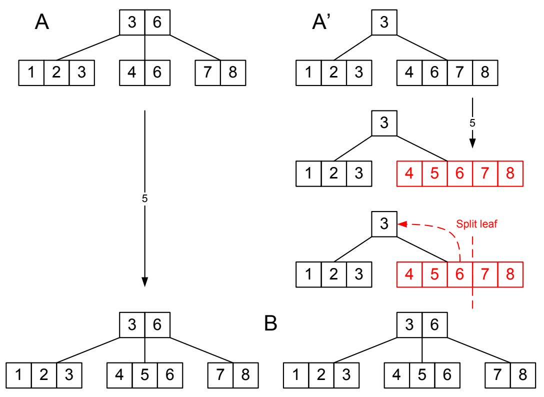

Fig. 3. $\mathcal{B}^+$-Tree resulting from two different $\mathcal{B}^+$-Trees.

of 3. Thus, the $\mathcal{B}^+$-Tree does not have much structure to analyze and the extraction of forensic evidence is difficult. We evaluated, based on a modified version of InnoDB how a more structured tree can be generated and what implications these modifications have on performance.

In InnoDB, both leaf and non-leaf nodes have a fixed size of 16 kilobytes. As non-leaf nodes of $\mathcal{B}^+$-Trees only store keys that have a typical length of a few bytes, there is enough space for adding thousands of keys to a node before it has to be split into two nodes. Thus, a typical non-leaf node in InnoDB has a huge number of children and the resulting tree mostly grows in width but not in height. To increase forensic value of a database tree, we considered two approaches that add structure to an InnoDB tree.

We first modified the fixed size of nodes in the source code of InnoDB. While node sizes other than 16 kilobytes are not officially supported, it is still possible to decrease it to a minimum value of 4 kilobytes. If we assume an integer primary key of 4 bytes, a non-leaf node in the original InnoDB storage engine can store up to 4,000 keys. We can reduce the value to 1,000 by defining a fixed node size of 4 kilobytes in the source code of InnoDB. This value, however, is still far too large for the generation of forensic-aware database trees. In a second evaluation, we increased the size of the primary keys in order to limit the number of keys that can be stored in one node. We defined a `varchar` primary key with a length of 767, which represents the maximum length of a key in InnoDB. Using the 450,000 test records from the previous evaluation, the resulting tree now has a branching level of 21, i.e., each node has up to 21 child nodes and the height of the tree raises to 7, which makes it more valuable for forensic investigations.

We compared performance of two InnoDB tables, one with a tree height of 3 (small primary key) and one with a height of 7 (large primary key). All experiments were performed with MySQL 5.1.44 on a machine equipped with an Intel Core i7 2.66 GHz CPU and 8GB of available system memory. The results (Table I) show that queries to the forensic-aware database are about twice as slow as to the database with the wide tree. While the performance losses are quite big, we argue that there exist cases (e.g. SOX compliant accounting) where security outranks performance.

TABLE I
PRACTICAL EVALUATION OF RUNTIME IMPACT.

Statement	tree height = 3	tree height = 7
Simple full table scan	0.0003s	0.0007s
Full table scan for single value	0.0055s	0.0115s
Update one column in entire table	0.1066s	0.2387s

IV. Conclusion and Future Research

In this work we outlined, how the intrinsic nature of a database's underlying $\mathcal{B}^+$-Tree can be utilized to thwart tampering by administrative personal able to forge the commonly used log mechanisms. Additionally, we discussed a special class of table-policies that can be used for developing audit tables that comply with regulations like the US Sarabanes-Oxley Act. For tables using this policy we are able to give strong forensic evidence for many cases of retroactive manipulation, thus providing the investigator with a new tool.

Regarding our future research concerning database-forensics, we aim at identifying other special subclasses of tables, for which strong forensic evidence can be provided by utilizing the $\mathcal{B}^+$-Tree-structure. Furthermore, we want to put a strong focus on qualitative analysis of the core problem: Identifying all forms of $\mathcal{B}^+$-Trees, for which strong forensic evidence can be provided and proofing the impossibility to do so for the rest. Additionally, we want to focus on the development of a practical tool that can be used by forensic investigators to compare database log files together with the $\mathcal{B}^+$-Tree of an old InnoDB-database-backup to the $\mathcal{B}^+$-Tree of the current database image.

References

[1] M. Mulazzani and E. Weippl, "Aktuelle Herausforderungen in der Datenbankforensik."

[2] B. Carrier, *File system forensic analysis*. Addison-Wesley Professional, 2005.

[3] C. Swenson, R. Phillips, and S. Shenoi, "File System Journal Forensics," *Advances in Digital Forensics III*, pp. 231–244, 2007.

[4] A. Burghardt and A. Feldman, "Using the HFS+ journal for deleted file recovery," *digital investigation*, vol. 5, pp. S76–S82, 2008.

[5] K. Eckstein, "Forensics for advanced UNIX file systems," in *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*. IEEE, 2005, pp. 377–385.

[6] K. Eckstein and M. Jahnke, "Data hiding in journaling file systems," in *Eingereicht beim Digital Forensic Research Workshop*. Citeseer, 2005.

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

[8] K. Pavlou and R. Snodgrass, "Forensic analysis of database tampering," *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 4, pp. 1–47, 2008.

[9] P. Fruehwirt, M. Huber, M. Mulazzani, and E. Weippl, "InnoDB Database Forensics," in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. IEEE, 2010, pp. 1028–1036.

[10] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Acta informatica*, vol. 1, no. 3, pp. 173–189, 1972.

[11] B. Ooi and K. Tan, "B-trees: bearing fruits of all kinds," in *Proceedings of the 13th Australasian database conference-Volume 5*. Australian Computer Society, Inc., 2002, pp. 13–20.