Using the Structure of B+-Trees for Enhancing Logging Mechanisms of Databases

Peter Kieseberg Sebastian Schrittwieser Lorcan Morgan
Martin Mulazzani Markus Huber Edgar Weippl

SBA Research
Vienna, Austria
[1stletterfirstname][lastname]@sba-research.org

ABSTRACT

Today's database management systems implement sophisticated access control mechanisms to prevent unauthorized access and modifications. This is, as an example, an important basic requirement for SOX (Sarbanes–Oxley Act) compliance, whereby every past transaction has to be traceable at any time. However, malicious database administrators may still be able to bypass the security mechanisms to make hidden modifications to the database.

In this paper we define a novel *signature* of a $\mathcal{B}^+$-Tree, a widely-used storage structure in database management systems, and propose its utilization for supporting the logging in databases. This additional logging mechanism is especially useful in combination with forensic techniques that directly target the underlying tree-structure of an index. The applicability of the approach is demonstrated by proposing techniques for applying this signature in the context of digital forensics on $\mathcal{B}^+$-Trees.

Categories and Subject Descriptors

H.2.0 [**Database Management**]: Security, integrity, and protection; E.1 [**Data Structures**]: Trees

General Terms

Security, Theory

Keywords

database forensics, b+ tree, database log

1. INTRODUCTION

$\mathcal{B}^+$-Trees are widely used in applications that have to handle large amounts of data such as database indexes or filesystems. Typically, database forensics is largely focused on extracting information from log files and checking data files for consistency (e.g.[1]). For example, in a recent work Frühwirt et al. [2] described the layout of the storage files used by InnoDB. Concerning filesystems, there have been several approaches for forensics on the file system layer ([3, 4, 5, 6]), still, these did not utilize the structure of the underlying $\mathcal{B}^+$-Tree. The only work known to us which describes reconstruction of the $\mathcal{B}$-Trees of filesystems in order to recover deleted files is by Koruga et al. [7].

In a previous paper [8] we proposed a method for utilizing intrinsic attributes of special tree structures of database indexes for forensic purposes. In the subsequent research, the question arose, how this approach could be enhanced, especially since it heavily relies on the existence of exact earlier versions of the $\mathcal{B}^+$-Tree. In order to provide for these, we introduce the *signature* of a $\mathcal{B}^+$-Tree which can be used for efficiently storing the structure of a $\mathcal{B}^+$-Tree and thus can be utilized as a basis for carrying out the forensic approach denoted in [8] and [9], as well as for direct forensic analysis. Our main contributions in this paper are the novel concept of a *tree signature* for $\mathcal{B}^+$-Trees that dramatically enhances the significance of forensic methods on the tree structure level, as well as two possible applications for enhancing database forensics on $\mathcal{B}^+$-Trees based on this signature.

2. BACKGROUND - DATABASE FORENSICS USING B+-TREES

In our previous work [8] we proposed using the structure of $\mathcal{B}^+$-Trees in order to detect manipulations of data entries by a malicious administrator. We outlined how to use the reconstruction of logged statements in the general case of arbitrary issued statements. Still, the approach lacked some feasibility in cases where an initial $\mathcal{B}^+$-Tree was missing. Therefore the work focused on tables with reduced instruction-sets, e.g. INSERT-statements only (or the dual case, reduced to DELETE-statements only) together with a strictly ascending (or descending) index. As a result we could show that the late insertion of additional records can be detected in certain cases.

This approach is reasonable for audit-tables, e.g. in large (and often complex) ticketing systems such as those used in telecommunication companies for billing and interconnection. In these instances, data arrives from switches (e.g. MSC, SMSC or Internet Gateways) and the processing of every record needs to be tracked in order to establish SOX-conformity (Sarabanes Oxley Act). Later inserts of additional tickets must not be possible in such systems, as well as in the underlying reference tables.

Concerning the general case, the main issue lies in the availability of a state prior to the manipulation of the tree in order to generate a comparable tree. This reduces the value of the approach since possibly large amounts of data must be stored for each intermediate tree. This is where our novel approach of tree signatures comes into play: By defining the *signature* of a $\mathcal{B}^+$-Tree, we are able to efficiently store the actual structure of a $\mathcal{B}^+$-Tree separately from the data. Changes in the data section can be retrieved by utilizing the transaction logs. Effectively, the use of our signature allows us to see INSERT-statements as bijective operations with respect to the underlying $\mathcal{B}^+$-Tree. Thus we are able to recalculate each single intermediate version of a tree easily by simple knowledge of the transactions applied to the database and the structural information in the signature log.

3. B+-TREE-SIGNATURE FOR LOGGING

3.1 Definition

We start by defining the *signature* of $\mathcal{B}^+$-Trees, since this is the prevalent data structure for database indices used in most modern relational database systems. Since the signature is only concerned with structure, we can easily adapt the definition to be suitable for $\mathcal{B}$-Trees and, as a starting point for future research, other balanced data structures. First, we define the *leaf signature*.

Definition Let $L_i, i = 1 \dots k$ be all leaf nodes of a $\mathcal{B}^+$-Tree B. Then we define the k-tuple $(|L_1|, \dots, |L_k|)$, where $|L_i|$ is the number of elements in node L_i, as the *leaf signature* of B.

The leaf signature already provides some valuable insights on the distribution of the data and could be used for digital forensics. Still, it is not sufficient for a unique description of the tree, since even with the (needed) knowledge of the data, several different trees possessing the same leaf signature (and thus the same partitioning of the data into leaf nodes) can be constructed. See Figure 1 for an example.

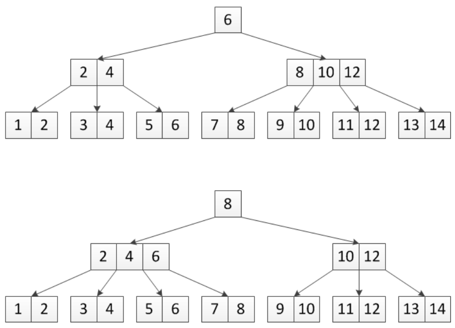

Figure 1: Two different $\mathcal{B}^+$-Trees with the same leaf signature

Definition Let $C_{i,j}$ be the jth node on the ith level of the tree, where nodes are counted in ascending order of the elements contained, and where $i = 1$ denotes the level of the root.

Let h be the height of the tree.

Let D_i be the *level signature*, such that $D_i = (|C_{i,1}|, \dots)$ for some i. Thus, D_i is the tuple containing the size of the nodes on the i-th level.

Let S be the *tree signature*, such that $S = (D_1, D_2, \dots, D_h)$. Thus, the *tree signature* is the tuple containing the level signatures of each level of the tree.

Using the definition above we will now prove that the information contained in the signature is sufficient for an exhaustive description of a $\mathcal{B}^+$-Tree.

Corollary 3.1. *The structure of a $\mathcal{B}^+$-Tree is well-defined by its signature.*

Proof. For our proof, we assume a non-empty tree with a valid tree signature. Let us consider D_1, the root level signature, which will always have the form (x). We know that the root of any valid tree will have $x + 1$ child nodes and, since we have assumed a valid tree signature, D_2 must therefore represent these $x + 1$ children. Since all elements inside a node are sorted in ascending order (by the definition of a tree), there is exactly one valid way of matching the elements of D_2 onto the children of the root of the tree.

Each child node can then be considered as the root of a subtree. By applying the same principle we can conclude that for each unique tree signature there is exactly one corresponding structure. Thus, the structure of a tree is well-defined by its signature. $\square$

Example Let B be a $\mathcal{B}^+$-Tree of order 4 containing the number $1 \dots 16$ with the following signature: $S = ((1), (2, 2), (3, 3, 2, 3, 3, 2))$. Then the resulting $\mathcal{B}^+$-Tree can be seen in Figure 2.

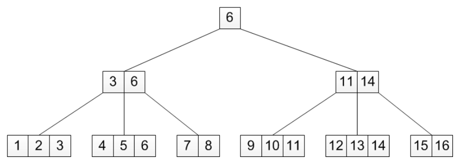

Figure 2: $\mathcal{B}^+$-Tree of order 4 with $S = ((1), (2, 2), (3, 3, 2, 3, 3, 2))$

3.2 Constructing trees from signatures

In this section we outline an algorithm for constructing a $\mathcal{B}^+$-Tree from the corresponding set of (sorted) index data together with the signature.

1. The last tuple of the signature is retrieved and the index data is partitioned accordingly, i.e. it is divided into blocks with the length derived from the last signature-tuple respectively. The resulting groups form the leaves of the $\mathcal{B}^+$-Tree.

2. The leaves are grouped according to the signature of the level above, i.e. by the penultimate tuple of the signature.

3. Step 2 is applied iteratively on each newly constructed level until we reach the root node (the first entry in the signature).

4. The final tree is constructed by promoting the correct elements into the inner nodes and the root node. This step is well defined by the $\mathcal{B}^+$-Tree-definition.

In the following, we give a short example in order to illustrate the algorithm.

Example Let B be the $\mathcal{B}^+$-Tree of order $b = 4$ as shown in Figure 2 together with the signature $S = \{(1), (2,2), (3,3,2,3,3,2)\}$.
In the first step of the algorithm we partition the data according to the last tuple of the signature in order to generate the leafs of the tree. The elements $1 \ldots 16$ partitioned by $(3,3,2,3,3,2)$ thus form the leaves $(1,2,3)$, $(4,5,6)$, $(7,8)$, $(9,10,11)$, $(12,13,14)$, $(15,16)$.
Now, each leaf node is an element in the next partitioning, i.e. the leaf nodes are grouped according to the signature of the next level $(2,2)$ thus resulting in two inner nodes with three child nodes each: $((1,2,3),(4,5,6),(7,8))$ and $((9,10,11),(12,13,14),(15,16))$. Combining these two subtrees with the root node yields to the final $\mathcal{B}^+$-Tree.

The complexity of the algorithm can be calculated in the following way: Since we assume that the n leaf-elements are already sorted in ascending order, partitioning can be done very fast (n operations). This has to be repeated for each level, where the maximum number of elements on each level can be estimated with $n_i \leq \log_{\frac{b}{2}+1} n_{i+1}$ (upper bound), thus resulting in the following formula for the whole tree (l denotes the number of levels of the $\mathcal{B}^+$-Tree.):

$$\sum_{i=l}^{1} A_i, \text{ with } A_i = \log_{\frac{b}{2}+1} A_{i+1}, A_l = n$$

4. APPLICATION

4.1 Additional log file

The signature log provides an additional log that can be used in order to detect manipulations of a database table. Since many operations result in changes of the underlying $\mathcal{B}^+$-Tree-structure, these operations will also affect the signature of the $\mathcal{B}^+$-Tree.

We also need to be aware of a malicious administrator who may be able to manipulate this log during forensic analysis. We now want to analyze the significance and robustness of this additional log in more detail.

In case the adversary is not able to tamper with the signature log (i.e. she does not hold the privileges or the ability to rewrite log files), the following information can be derived:

1. The structure of the current tree must correspond to the signature in the signature log.

2. The number of logged signatures must be the same as operations in the transaction log, so we can use the signature log to cross-check for any obvious manipulation in those transaction-logs.

Tampering with the signature log is far from trivial, since:

- The adversary must be able to read out the structure of the $\mathcal{B}^+$-Tree by herself.

- Insert very subtle changes, since even small changes in the insert order of elements can have a great impact on the structure of the resulting trees - especially when there are more complex transactions following the one that got changed [8].

- In the end, all these changes must lead to a tree that has the same structure like the one that is observed at the moment.

As was already discussed in Section 2, the structure of the underlying $\mathcal{B}^+$-Tree can be used for forensic analysis. In particular ordered insert statements can be revealing.

4.2 Reconstructing old Tree-versions

As shown in our previous work, just logging the transactions issued against a database is not enough to preserve a log of all previous structures of the underlying $\mathcal{B}^+$-Tree, since in general the operation of inserting an element into a $\mathcal{B}^+$-Tree is not bijective (the inverse operation is not injective). Moreover, when dealing with multiple INSERT- and DELETE-operations, it is usually impossible to reconstruct the former tree.

Example The example 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.

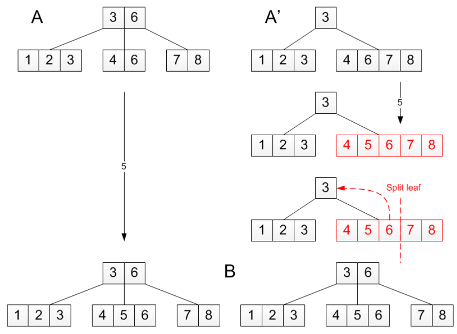

Figure 3: $\mathcal{B}^+$-Tree resulting from two different $\mathcal{B}^+$-Trees

Since the signature of the tree (together with its characterization by its order) identifies the structure, the signature log together with the tree and a log of all operations on the tree can be used to retrieve any intermediate state, since:

1. The partitioning of the nodes (and thus the structure) of the $\mathcal{B}^+$-Tree is denoted in the signature,

2. The data inside the $\mathcal{B}^+$-Tree can be retrieved by combining the current tree with the information written in the change logs for every intermediate state,

3. The combination of this information (together with the basic characterization of the tree itself) defines the whole tree.

Thus, the original tree can be restored with a combination of the data (taken from other log files) and the signature log. Again, we can apply the algorithm described in Section 3.2.

1. Be E the set of all elements in the current $\mathcal{B}^+$-Tree.

2. Choose a point in time t, for which the tree should be reconstructed.

3. Extract $D(t)$ and $I(t)$ from the transaction logs, where $D(t)$ is the set of all deleted elements and $I(t)$ the set of inserted elements since t.

4. Join E with D and subtract I in order to calculate E(t), the set of all elements at point in time t: $E(t) = (E \cup D(t)) \setminus I(t)$.

5. Use the tree signature $S(t)$ at point in time t and the algorithm proposed in Section 3.2 to reconstruct the old state of the $\mathcal{B}^+$-Tree.

Obviously this approach is also invulnerable against reorganizations of the $\mathcal{B}^+$-Tree due to performance optimization strategies, since these operations only change the underlying structure of the $\mathcal{B}^+$-Tree, which is already stored in the signature log.

One additional prerequisite must be fulfilled though: UPDATE-statements must be logged as a combination of the respective INSERT- and DELETE-statements.

Example Let $E = \{1, 2, 3, 4, 5, 6, 7, 8\}$ be the elements of a $\mathcal{B}^+$-Tree of order 4 that was constructed from a previous $\mathcal{B}^+$-Tree with signature $S(t) = ((2), (2, 2, 3))$ by inserting the elements 2 and 4, as well as deleting element 9.
Thus we can construct the old tree with $E(t) = ((E \cup D(t)) \setminus I(t) = \{1, 3, 5, 6, 7, 8, 9\}$ where $D(t) = \{9\}$ and $I(t) = \{2, 4\}$ and reconstruct the $\mathcal{B}^+$-Tree with respect to the signature $S(t)$. The results can be seen in Figure 4.

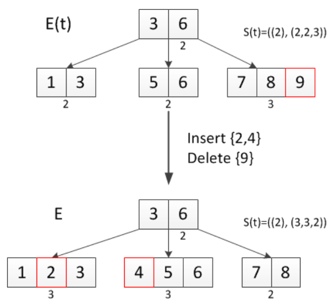

Figure 4: Reconstructed $\mathcal{B}^+$-Tree

In case of differences between a reconstructed tree and an old backup, manipulations can be detected.

4.3 Limitations

One limitation of this approach lies in the fact that the technique must be implemented inside the DBMS, i.e. it is possible to do this via external tools, but to us this seems a rather costly alternative concerning performance. So the vendor of the DBMS must be convinced to change the software, which is rather unlikely concerning the big players in this market.

Another limitation stems from the issue that the bijectivity of the transaction can only be assured when *every* change in the underlying structure is logged. This will result in some additional space needed, as well as an additional logging operation after each transaction. Since every log entry covers the structure of the whole tree, this could result in mitigation strategies for this problem, including:

- In case a transaction only affects a sub-tree, the logging-logic could reuse the last entry of the signature log and directly alter the part of the signature that is concerned with this sub tree.
- The logging could rely on some external tool that calculates intermediate steps out of two tree signatures (extracted the usual way) and the transaction log.

The most important drawback, however, lies in the fact that an almighty administrator is, at least in theory, able to completely fake the log. Even though, faking the signature log should be drastically more complicated compared to a transaction-log, this possible threat has to be taken into consideration.

5. CONCLUSION

In the course of this paper we proposed the *signature* of a $\mathcal{B}^+$-Tree, a set of tuples describing the structure of an index and a logging mechanism for databases based on this signature. We showed that this mechanism is useful for forensic analysis applied to the underlying $\mathcal{B}^+$-Tree-structure of an index. With this signature mechanism, database transactions become bijective with respect to the structure of the actual index tree, thus allowing for a multitude of forensic approaches. Additionally we believe that this mechanism can be very useful in the case of defining a forensic-aware database, since even in case of index reorganizations it is possible to successfully reconstruct old versions.

6. ACKNOWLEDGEMENTS

This work has been supported by the Austrian Research Promotion Agency under grant 824709 (Kiras).

7. REFERENCES

[1] K. Pavlou and R. Snodgrass, “Forensic analysis of database tampering,” *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 4, pp. 1–47, 2008.
[2] 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.
[3] B. Carrier, *File system forensic analysis.* Addison-Wesley Professional, 2005.
[4] C. Swenson, R. Phillips, and S. Shenoi, “File System Journal Forensics,” *Advances in Digital Forensics III*, pp. 231–244, 2007.
[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 *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] P. Kieseberg, S. Schrittwieser, M. Mulazzani, M. Huber, and E. Weippl, “Trees cannot lie: Using data structures for forensics purposes,” in *European Intelligence and Security Informatics Conference (EISIC 2011)*, 9 2011.
[9] M. Mulazzani and E. Weippl, “Aktuelle Herausforderungen in der Datenbankforensik.”