# Indexing large updatable Datasets in Multi-Version Database Management Systems

Christian Riegger
Data Management Lab
Reutlingen University
Germany
christian.riegger@
reutlingen-university.de

Tobias Vinçon
Data Management Lab
Reutlingen University
Germany
tobias.vincon@
reutlingen-university.de

Ilia Petrov
Data Management Lab
Reutlingen University
Germany
ilia.petrov@
reutlingen-university.de

**Abstract**

Database Management Systems (*DBMS*) need to handle large updatable datasets under on-line transaction processing (OLTP) workloads. Most modern *DBMS* provide snapshots of data in multi-version concurrency control (*MVCC*) transaction management scheme. Each transaction operates on a snapshot of the database. It is calculated from a set of tuple versions, containing logical transaction timestamps to determine whether a version belongs to the snapshot. This transaction management scheme enables high parallelism and resource-efficient append-only data placement on secondary storage. *One major issue in indexing tuple versions on modern hardware technologies is the high write amplification for tree-indexes.*

Partitioned B-Trees (*PBT*) [5] is based on the structure and algorithms of the ubiquitous B$^+$-Tree [8]. They achieve a near optimal write amplification and beneficial sequential writes on secondary storage. Yet they have not been implemented in a *MVCC* enabled *DBMS* to date.

In this paper we present the implementation of *PBT*s in *PostgreSQL* extended with *SIAS*. Compared to PostgreSQL's standard B$^+$–Trees *PBT*s have 50% better transactional throughput under TPC-C and a 30% throughput improvement to standard *PostgreSQL* with *Heap-Only Tuples*.

**Keywords:** Indexing Structure, MVCC, Modern Storage Hardware

**Topics:** Information Systems, Database Management Systems, Data Structures, Data Access Methods

## 1 Introduction

In times of Big Data, IoT, cloud computing, blockchain and social media, datasets are large and and update-intensive. Database Management Systems (DBMS) and Key/Value Stores

(KV-Stores) are predestined to manage these datasets, but are the bottleneck of most data-intensive operations. Datasets have a near-linear growth and cannot be entirely located in main memory in most cases.

Whenever a transaction modifies a tuple in a multi version concurrency control (MVCC) enabled DBMS such as PostgreSQL, a new version-record (tuple version or simply version) of this tuple is produced. Existing approaches in such DBMS organize versions as a doubly-linked list, where each version record has a *creation_timestamp* and an *invalidation_timestamp*, which is initially empty. Whenever a transaction $TX_n$ creates successor version upon an update, both version records are modified, setting the *invalidation_timestamp* of the predecessor, and the *creation_timestamp* of the successor to *timestamp($TX_n$)*. We consider a novel version-organization, based on SIAS [3]. Every version has a *creation_timestamp*, and a single backward reference to its predecessor.

This version model assumes that every tuple comprises a set of tuple versions that are available as persistent version records, physically stored as a chain. While processing a query under a transaction, only the "visible" versions should be determined and passed on for processing. The Snapshot Isolation visibility criteria hold, i.e. a version $t_x.v_y$ is visible to transaction $TX_4Q$, if:

(1) $creation\_timestamp(t_x.v_y) = MAX(creation\_timestamp(t_x.v_{\{ALL\}})) < timestamp(TX_4Q)$;

(2) $transaction\_status(creation\_timestamp(t_x.v_y)) = COMMITTED$; and

(3) $creation\_timestamp(t_x.v_y) \notin L_{concurrent}(TX_4Q)$.

Hence the version visibility check is very I/O intensive.

With this model, searching for one or a few data tuples with specific search predicates in base tables is an expensive operation with super-linear growth. In times of Big Data, when datasets typically cannot be entirely located in main memory, full table scans are not an option. *Indexes* describe an additional access path to data tuples located in base tables. The index structure of a B$^+$-Tree [8] became ubiquitous in database management systems [7]. The tree-index allows accessing data in a key-sorted order in logarithmic time. Index record maintenance operations in the sorted tree-structure, as well as resulting structure maintenance operations cause a high write amplification to secondary storage – this effect is amplified by the maintenance of tuple versions. *Indexing tuple versions is still an open research area, considering characteristics of modern storage hardware. Index structures need to handle modifications of index records out-of place for reduction of write amplification on secondary storage media.*

Partitioned B-Trees ($PBT$) [5] is based on the structure and algorithms of the ubiquitous B$^+$-Tree [8]. They achieve a near optimal write amplification and beneficial sequential writes on secondary storage by collecting modifications in a main memory partition and forcing related nodes to secondary storage at eviction time. Already persisted data is not physically affected

by further modifications – i.e. maintenance of tuple versions in the tree-index does not amplify write amplification.

In this paper we present the implementation of *PBT*s in PostgreSQL extended with the version model of SIAS. Compared to PostgreSQL's standard B$^+$-Trees *PBT*s achieve a 50% improved transaction throughput under TPC-C and a 30% throughput improvement to standard *PostgreSQL* with *Heap-Only Tuples*.

The structure of this paper is as follows. We give an overview of related indexing approaches in Section 2 and outline the characteristics of modern hardware technologies in Section 3. In Section 4 we discuss the conflict in design decisions of MVCC. We outline the algorithms of PBT in Section 5 and verify our assumptions in Section 6.

## 2 Related Work

Most popular indexing approaches in database management systems (DBMS) are based on B$^+$-Trees, which can result in high write amplification on random updates in large datasets. PostgreSQL uses Heap-Only Tuples (HOT) as indirection layer to reduce index management operations. Index records reference items in base table, which point to tuple versions in the heap node. Corresponding tuple versions are hold on the same table node and can be located by processing the version chain. If a tuple version becomes garbage collected, the item is modified to reference the next relevant version. This indirection layer reduces index modifications, but cannot avoid write amplification of index nodes. Furthermore the write amplification of base table nodes is increased for large datasets.

Maintaining out-of place tuple versions enable high parallelism and a beneficial append-only sequential write pattern to secondary storage for base tables. Snapshot Isolation Append Storage (SIAS) makes use of the natural append-only characteristics of tuple versions and achieved an increased throughput of 30% in comparison to PostgreSQLs standard base table organization in a TPC-C like on-line transaction processing (OLTP) workload [3]. Every version has a *creation_timestamp*, and a single backward reference to its predecessor. This version model assumes that every tuple comprises a set of tuple versions that are available as persistent version records, physically stored as a chain. However, this version model does not support HOT indirection layer, whereby indexing effort is increased.

### 2.1 MV-IDX

MV-IDX [4] is based on a B$^+$-Tree and maintains a virtual identifier for each tuple and in-memory data nodes for each version as an indirection layer. With Snapshot Isolation Append Storage (SIAS) [3] write amplification on base tables is reduced in comparison to HOT, but index management operations can cause in-place updates and a high write amplification – e.g. if an indexed attribute value becomes modified. Partitioned B-Trees (PBT) handle updates to indexed attribute values in a main memory partition in PBT-Buffer, whereby write

amplification is optimized.

### 2.2 Write Optimized B-Trees

The Write Optimized B-Tree [6] aims to achieve a log-structured write pattern in a $B^+$-Tree with transactional support. It is organized like a traditional $B^+$-Tree with limited modifications. Traditional $B^+$-Trees use sibling pointers to chain nodes at leaf level or at each level. These are used to support scans in ascending and descending order and for further reasons. Another pointer is maintained at separator keys in parent nodes to find records in leaf nodes in a traversal operation. If a page changes its location, e.g. for achieving a log-structured write pattern, three pointers change and so three further pages have to be modified. Write Optimized B-Trees switch hard coded sibling pointers to so called fence keys. Fence keys describe minimum and maximum key, a page contains. They can have a exclusive or inclusive meaning. As a consequence, page locations can change for log-structured writes, without changing sibling pointers at neighboring pages. Nonetheless, the pointer at separator key in parent page has to change. Doing so, it enables a log-structured write pattern, but is also able to support traditional $B^+$Tree like update in-place operations, for example, if disk space is rare. Missing sibling pointers offer a new problem. Cursors on scans cannot find siblings with fence keys. Therefore, the next separator key in parent page has to be requested. As a result, additional complexity in cache management occurs to hold parent pages.

Write Optimized B-Trees do not solve the problem of high write amplification in $B^+$-Trees. If a page gets evicted, it is written in a log-structured manner. However, a page is not protected from further modifications and already indexed data has to be written manifold. Partitioned B-Trees (PBT) collect modifications to a leaf node in PBT-Buffer until the whole partition gets evicted. Every record is written exactly once, except for further reorganization and garbage collection – write amplification is near optimal.

### 2.3 LSM-Trees

LSM-Trees [9] are optimized for high update rates and reduce write amplification due to collecting and pre-sorting modifications in a fixed-sized main memory component, which becomes evicted on a certain threshold and replaced by a new main memory component. As a result, several components exist on secondary storage media and are frequently merged in larger components to improve look-up and scan performance. Pre-sorted records are migrated and sequentially written in a log-based manner. bLSM-Trees [10] are based on the structure of LSM-Trees, however, there is a fixed count of three components for reduction of read amplification. Furthermore, bloom filters protect components from unnecessary reads for point queries. Scheduling of merge areas and insertion rates between components reduce steals and replacement selection increases the effective amount of merged records. Each component has the structure of a $B^+$-Tree. Due to the fact, that the capacity is in a logarithmic relation to the height of a tree, more inner nodes are required than in a single tree structure. This increases

read amplification to secondary storage, reduces cache hit rates and shrinks performance in scans.

Advantages of Partitioned B-Trees (PBT) are manifold. *First*, the single tree-index structure leverages the logarithmic relation between capacity and height of the tree. Index nodes are commonly used and buffered across partitions, whereby cache hit rate is increased at same height like larger components in LSM-Trees. *Second*, compression methods, like suffix truncation, perform better in one large set of records, than in several smaller sets [2]. *Third*, partition sizes are self-balanced and workload adaptive due to commonly used PBT-Buffer. Partitions in update-intensive PBTs are automatically prioritized. Managing component thresholds in LSM-Trees requires more knowledge about the workload and administrative effort. *Fourth*, partitions of PBTs are more flexible than components in LSM-Trees. A partition can be created to absorb bulk loads with low effect on concurrent workload. Afterwards, the partition can be merged in main memory, evicted to secondary storage or cropped from the tree structure, based on the result of the transaction. Furthermore, *Cached Partitions* can be similar created out of result sets of frequently queried records. Lower numbered partitions can be skipped for look-ups of records in *Cached Partitions*. On workload switches they can be cropped from tree structure or evicted to further storage media in memory hierarchy. Components in LSM-Trees are not designed for additional complexity. *Last*, partitions in PBTs are immutable once they have been sequentially evicted to secondary storage media. This behavior reduces write amplification and enables well-sized efficient filter structures for minimizing read amplification. Reorganizations and garbage collection is flexibly performed for several partitions when needed. LSM-Trees frequently merge and migrate records to larger components, whereby write amplification is increased.

# 3 Characteristics of modern Hardware Technologies

Several trends in hardware technologies lead to re-thinking traditional architectures and algorithms. Developments in computing and storage technologies as well as increased main memory volumes require reflections of an optimal usage of layers in memory hierarchy and parallel computing concepts.

Novel semiconductor-based storage technologies close the access gap in memory hierarchy between fast Random Access Memories (RAM) and slow Hard Disk Drives (HDDs) – Non-Volatile Memories (NVM) and Solid State Drives (SSD). Both are able to operate as persistent secondary storage media, like HDDs, but distinguish in most other characteristics. *First*, they have shorter latencies, especially for random read access. *Second*, there is an asymmetry in read and write latencies. Reads are ten times faster than writes and a hundred times faster than erases. *Third*, updates to pages are performed out-of place. This is realized by an insertion of the modified page and an invalidation of the original one. Several pages are connected to

blocks. If pages in a block were invalidated, the still valid pages are copied in a new block and the old one becomes erased as part of the garbage collection. *Fourth*, high parallelism in SSDs support sequential writes – latencies depend on the access pattern of a SSD.

Moreover, sequential writes of a couple of pages enable a SSD to fill a block with related pages to a file. If the file or a part of it would be removed, the whole block can be erased without copying still valid pages. Mayor disadvantage of a SSD is its durability. Every erase and write degenerates a cell. Lifetime of a SSD shortens by every write and logical update operation. Wear leveling methods aim to distribute writes to every cell and extend lifetime of a SSD, but increase its inner write amplification.

Established architectures and algorithms have to be revised for characteristics of modern hardware. $B^+$-Tree is an established index structure. Modern hardware can handle its read behavior very well due to short read latencies. Writes in a random manner and in-place update operations are to disfavor, considering performance and lifetime of SSDs. Nonrecurring sequential writes of related pages are preferable. An increase of read operations is negligible due to short read latencies and high parallelism of SSDs. Furthermore, algorithms have to process in parallel and caches have to be filled to take effort of new computing technologies. The Partitioned B-Tree takes effort of these developments.

# 4 Multi-Version Concurrency Control

Multi-version concurrency control (MVCC) is the most popular transaction management scheme in modern database management systems (DBMS). For instance, it is used by Oracle, MySQL-InnoDB, HyPer, SAP HANA, MongoDB-WiredTiger, NuoDB and PostgreSQL. It is also possible to enable MVCC in Microsoft SQL-Server. In theory, MVCC enables high parallelism, because reading transactions do not block concurrently writing transactions. Modifications result in a new tuple version. Furthermore, in snapshot isolation, modifications by writing transactions do not block concurrently reading transactions, because for each transaction a visible tuple version can be returned.

The DBMS implement MVCC transaction management scheme in different ways. Differences in design decisions can be found in the *(a) Concurrency Control Protocol*, *(b) Version Storage*, *(c) Version Ordering*, *(d) Garbage Collection* and *(e) Index Management* [11]. In fact, that *(a) Concurrency Control Protocols* deal with serialization strategies (first updater / committer wins) and has low effect on indexing and resulting write and read I/O patterns on secondary storage, we focus on and outline points *(b)* to *(e)* in the following. Afterwards, we give a short discussion. There is a conflict dilemma in usage of the optimal design decisions for large datasets and characteristics of modern storage technologies (outlined in Section 3).

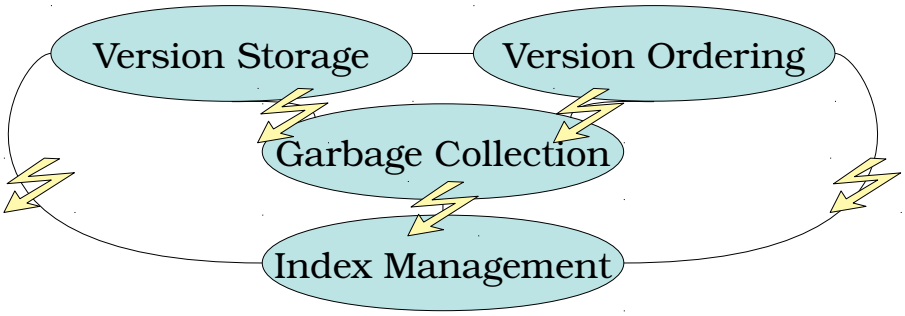



Figure 1: Dilemma: Conflicting characteristics of Version Storage, Version Ordering, Garbage Collection and Index Management

### 4.1 Version Storage

Tuple versions correspond to one logical tuple. They form a linked list, which represents a version chain. Mainly, a version of a tuple can be maintained in two different ways – logical or physical. The first type means that for each modification of a logical tuple a delta record indicates the difference to another version. These delta records are connected and required to restore the tuple version. Physical storage means that each tuple version is entirely stored.

In both cases following information is required: a *content / delta* that is stored in the DBMS for a tuple, logical *timestamps* (transaction numbers) for *validation* and *invalidation*, and a *reference* to its *predecessor* and / or *successor version* – based on the version ordering. Storing a delta with logical version maintenance can reduce the required size for large tuple contents, if only one or a few attributes change on modifications, but restoring the physical tuple version requires additional sequential processing. In the case of physical version maintenance, the content has not to be restored, but is likely to require more space.

Modifications are performed in-place or out-of-place. Logical version maintenance in-place means that the already stored tuple is modified to the information (contents, logical timestamps) of the new tuple version afterwards the old one was inserted as a delta record somewhere else, e.g. in a log. Out-of-place updates would result in a simple insertion of a delta record referencing to its predecessor, e.g. a further delta record. In-place updates of physical version maintenance result in storing a copy of the current tuple version in another location, e.g. a temporal table, and like in case of logical version maintenance the modification of the already stored tuple. Out-of-place updates of physical version maintenance means the insertion of a new tuple in the base table. Based on the version ordering, may additional modifications are required, e.g. logical timestamps or references.

Considering the characteristics of modern storage technologies, out-of-place updates are preferable due to less write amplification to secondary storage for large datasets. Furthermore, this behavior enables higher parallelism than in-place updates, whereby the tuple version has

to be exclusively latched for modification. This is possible with logical and physical version storage. Delta records tend to consume less space than physical tuple versions, but require additional processing and all predecessors or successors for tuple reconstruction.

### 4.2 Version Ordering

As already mentioned, a version chain of a logical tuple forms a linked list. A doubly linked list enables knowledge of predecessor and successor, however, it requires additional latches and reduce parallelism on modification [11] in comparison to singly linked lists, which require a version ordering. Discovering the visible tuple version to a transaction snapshot requires to follow the version chain from an entry point, until the matching version was found. Basically, there are two different ordering methods – old-to-new (O2N) and new-to-old (N2O) for singly linked lists.

For both methods also in-place and out-of-place updates are possible. In case of O2N-ordering, the entry point is the oldest tuple version in version chain. A visibility check requires to process all predecessors, beginning from the oldest tuple version. This behavior might be good for look-ups of long-lasting analytical queries, where older tuple versions are likely to be visible to a transaction, but OLTP workloads mostly require a recent version and have to process the whole version chain. Modifying the tuple means for the out-of-place method an update of the invalidation timestamp and the reference of the old tuple version afterwards an insertion of the new tuple version. In-place updates would result in a copy of the old version (with modifications of the invalidation timestamp and reference to its origin) and an in-place modification of the tuple content and validation timestamp.

N2O-ordering means that the entry point is the most recent tuple version, which references to its predecessor. Queries in OLTP transactions can find the visible version very well, because the most recent tuple version is the entry point of the version chain. In-place updates require an exclusive latch, a copy of the old tuple version and a modification of its invalidation timestamp as well as modifications at its origin for the new tuple version (content, validation timestamp, reference). Out-of-place version storage only requires an insertion of the new tuple version with a reference to its predecessor. Its validation timestamp is equal to its predecessors invalidation timestamp. In fact of processing the version chain from the entry point, invalidation timestamps of predecessors are known and do not have to be modified.

Considering the characteristics of modern storage technologies new-to-old (N2O) version ordering for physical version storage result in best write amplification and with append-only characteristic for large datasets. All other approaches require in-place updates, which shrink the benefits in parallelism of a singly linked list and are persisted to secondary storage at some point in time.

### 4.3 Garbage Collection

Tuples are modified multiple times. In MVCC modifications result in new tuple versions. If no time travel functionality is required, old tuple versions become obsolete, if they are no more visible for any active transaction. Garbage collection (GC) reclaims space and can improve performance, especially for O2N version ordering. However, GC increases write amplification on secondary storage.

Garbage collection (GC) can be performed on transaction and tuple level. For transaction level GC, the DBMS keeps track of read and write sets of any transaction. Obsolete predecessor tuple versions are detected by the write sets of transactions and become garbage collected. Tuple level GC can be performed in two different ways – background vacuum and cooperative cleaning. In the first case, a background thread scans data and purges obsolete versions. Cooperative cleaning uses the process operation of version chains for detection of obsolete tuple versions. This approach works well with O2N version ordering. The drawback is that not processed versions are not identified and a background vacuum cleaner is required, so this approach normally does not work with N2O version ordering. [11]

Garbage collection (GC) is difficult, considering characteristics of modern storage technologies. In the best case, GC can be performed in main memory, but for large datasets, that is not always possible. Modern semiconductor based secondary storage technologies prefer a sequential append-only write pattern. As a result GC should be performed for large regions of obsolete tuple versions to reduce write amplification of still required ones. Furthermore, if the entry point of the version chain becomes garbage collected, first references in additional access paths have to be updated. GC would affect several data structures in parallel and cannot be separately performed. In N2O version ordering, the indexed entry point is the most recent tuple version, which is likely to be not garbage collected. Indexes can perform GC independent from base tables.

### 4.4 Index Management

Complexity of index management strongly depends on version storage and ordering techniques. A lossy result from index look-ups and scans is not acceptable, so in theory every tuple version should be indexed. This approach can result in massive write amplification on secondary storage, e.g. in case of $B^+$-Trees. Furthermore, mostly used indexes do not support visibility checks in MVCC transaction management schemes. Therefore, the version chain of the base table is required to determine the tuple version, which is visible to a transactions snapshot. As a result, at least the entry point of the version chain must be indexed. Moreover, modifications in the tuple versions content, which affect a search key column of an index record have to become visible to an unlossy index.

There are two possibilities to map index records to tuple versions in base tables. First, physical references can be used. With this method, the entry point tuple version in base tables can be directly accessed, but changes to the entry point location result in modifications of the

index record. Second, an indirection layer with logical references is implemented. Therefore, each tuple version of a tuple is referenced with an unique identifier. Index records reference to this unique identifier in the indirection layer, which references the entry point tuple versions physical storage location. This approach reduces index modifications, if the entry point changes, but requires additional structures and processing.

In-place updates of tuple versions in base tables reduce index maintenance operations with physical indirection. However, as outlined in Sections 4.1 and 4.2, an out-of-place append-only scheme brings benefits on modern storage technologies. Indexing tuple versions with unique identifiers and an indirection layer can reduce index maintenance costs for the preferred storage scheme, if the search key attributes of the tuple content do not change, but requires additional structures and processing for look-ups. However, inserting and modifying tuples in a traditional strict alphanumeric-sorted index structure, like a $B^+$-Tree, result in in-place updates on index nodes and high write amplification.

### 4.5 Discussion

We have outlined most relevant design decisions for storing tuple versions in multi-version concurrency control (MVCC) transaction management scheme. Every design brings benefits for specific tasks and requirements. We focus on large update-intensive datasets, which cannot be entirely located in main memory. Therefore, we introduced the dilemma in different design decisions.

Modifications are preferably stored as physical tuple versions in base tables due to tuple reconstruction costs. Out-of-place updates reduce write amplification to secondary storage. This can be achieved by a new-to-old (N2O) version ordering, because invalidation timestamps of predecessors can be reconstructed from previously processed successors creation timestamps and do not have to be maintained in a latch-free singly linked list. Garbage collection (GC) is required for space reclamation, but brings additional complexity to data structures. This process should be performed in main memory or for large regions of invalidated tuple versions. Base tables and additional access paths should be able to perform GC independent from each other.

A N2O version ordering requires index maintenance for every new tuple version, because the entry point of the version chain changes. A logical indirection layer could reduce index maintenance effort, however, a sequential write pattern to secondary storage cannot be achieved with traditional indexing structures. Furthermore, an indirection layer brings additional complexity to GC and processing costs for visibility check in the version chain. Due to high update rates to indexes, caused by insertion of new tuple versions on modifications, traditional index structures become a bottleneck.

We decided to implement Partitioned B-Trees (PBT) in PostgreSQL extended with Snapshot Isolation Append Storage (SIAS)[3], due to its beneficial append-only write I/O properties to secondary storage. It fits well with the outlined conclusions for tuple version storage in base

tables. We describe the structure and algorithms of PBT and how it is able to achieve the preferable sequential write pattern to secondary storage.

# 5 Approach: Partitioned B-Trees

Partitioned B-Trees (PBT) [5] are based on traditional $B^+$-Trees [8] and make use of its intrinsic and well studied algorithms with few modifications. The essential difference is an introduced artificial leading key column – the partition number. An index record consists of a *partition number*, its *search key columns* and a *physical tuple reference* or an *unique virtual identifier* for tuple assignment. Every different partition number value describes a single partition. Therefore, a partition number can be an integer value of two or four bytes. This enables the PBT to maintain partitions within one single tree structure in alphanumeric sort order. Partition numbers are transparent for higher database layers and each PBT maintains partitions independent of other PBTs. Partitions appear and vanish as simple as inserting or deleting records and can be reorganized and optimized on-line in server-transaction merge steps, depending on different workloads. Partitions can support additional functionalities, like bulk loads or can serve as multi-version store [5].

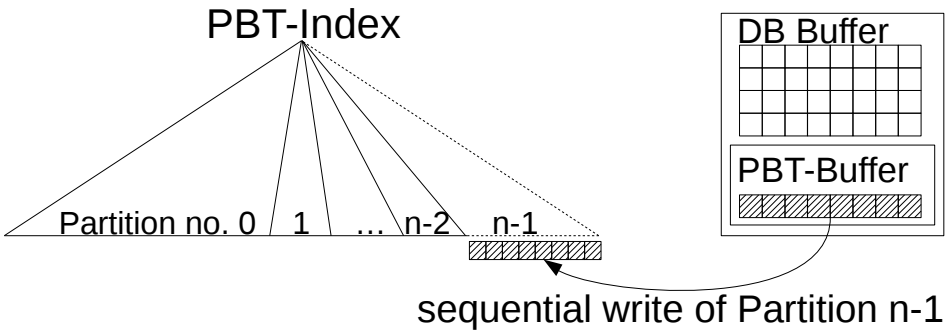



Figure 2: Sequential write of a Partition

PBTs write any modification of index records exact one time on eviction of a partition, except for later reorganization or garbage collection operations, what enables a beneficial sequential write pattern to secondary storage. This is realized by forcing sequential writes of all leaf nodes of a whole partition. The operation is illustrated in Figure 2. Leaf nodes of modifiable main memory partitions are stored in a separate area of the buffer cache – the PBT-Buffer. This area is shared for all PBT indexes in database schema. Records can be inserted or updated only in partitions, which are located in the PBT-Buffer. In case of full PBT-Buffer, one PBT is selected and one of its main memory partition is written to secondary storage. A cache logic decides, which partition has to be evicted. A write operation has to clean up as much space as possible, but partitions should be able to grow. We decided to evict the largest partition.

*First*, a new partition is created to support ongoing modifications, if required, and the partition, which has to be evicted, becomes immutable. *Second*, a bloom filter and prefix bloom filter is created, gets filled with all index records in the recently closed partition and is flushed to secondary storage. *Last*, all leaf nodes are sequentially written to secondary storage (see Algorithm 1).

**Algorithm 1** Partitioned B-Tree - EVICT

**Input:** Partition $p_{evict}$ selected by the cache logic
1: **procedure** EVICT( $p_{evict}$)
2: Let $filter \leftarrow$ new $(prefix)bloomfilter$ that can manage $p_{evict}.record\_count$
3: add $filter$ to $p_{evict}$
4: Add new $Partition$ to B$^+$-Tree $PartitionsList$
5: Set $p_{evict}$ immutable
6: **for each** $index\_record \in leaf\_nodes$ of $p_{evict}$ **do**
7: $filter$.ADD($index\_record.key$)
8: **end for**
9: FLUSH($bf$)
10: CLEAN( ) visited $leaf\_nodes$
11: **end procedure**

PBT indexes in MVCC transaction management scheme are not lossy, however, they are only able to return a set of entry points to candidate tuples, which have to be verified in a visibility check. Therefore, they have to perform index updates, if the entry point changes or search key columns are modified by the contents of a new tuple version in base table. We describe the index operations in a PBT:

**Insert Operations** are only performed in a mutable main memory partition. Therefore, the first search key column is prepended with its partition number. Physical tuple reference and unique virtual identifier to the tuple version in base table are maintained. The index structure is traversed and the index record is inserted at its regular position in the B$^+$-Tree structure. Remember, the alphanumeric sort order is affected the partition number, so the leaf node is guaranteed to be in main memory. Uniqueness constraints are supported by first performing a read operation.

**Algorithm 2** Partitioned B-Tree - INSERT

**Input:** Regular $|attr_{val}|, reference$
**Output:** $ErrCode$
1: **procedure** INSERT($|attr_{val}|, reference$)
2: Let $partition_{insert} \leftarrow$ MAX($PartitionsList$)
3: Let $partitioned\_record \leftarrow$ FORM_PARTITIONED_RECORD($partition_{insert}, |attr_{val}|, reference$)
4: UNIQUENESS_CONSTRAINT_CHECK($|attr_{val}|$) ▷ check all $Partitions$
5: **return** DO_REGULAR_INSERT($partitioned\_record$)
6: **end procedure**

**Update Operations** can be performed in-place, if the index record is still in a mutable main memory partition. Therefore, only the physical tuple reference field has to be modified to the new tuple version in new-to-old (N2O) version storage, if modifications to the tuple content does not affect search key columns in the index. If the index record of the updated tuple is in an evicted immutable partition on secondary storage or search key columns are affected, an insert in a mutable main memory partition is performed. In case of using an indirection layer, only updates to the search key columns are required in the index structure. In the other cases, an update to the indirection layer is sufficient.

**Algorithm 3** Partitioned B-Tree - UPDATE

**Input:** Regular $|attr_{val,old}|, |attr_{val,new}|, reference$
**Output:** $ErrCode$

```
1: procedure UPDATE(|attr_val,old|, |attr_val,new|, reference)        ▷ update w/ physical reference
2:     Let partition_update ← MAX(PartitionsList)
3:     Let partitioned_record ← FORM_PARTITIONED_RECORD(partition_update, |attr_val,old|, reference)
4:     if |attr_val,old| = |attr_val,new| and FIND(partitioned_record) then
5:         return in_place_update(|attr_val,old|, reference)        ▷ update physical reference
6:     else
7:         return INSERT(|attr_val,new|, reference)
8:     end if
9: end procedure
```

**Delete Operations** are performed similar to update operations. The physical tuple reference should point to a tombstone record maintained in base table. If an indirection layer is used, an update to its entry point is sufficient.

**Search and Scan Operations** are not allowed to be lossy, however, they return a set of candidate tuples, which have to be verified in a visibility check in base table. The query search predicates are modified to match the search key columns in a PBT – a partition number is prepended to the first search key column. The partitions in a PBT are traversed and scanned from the highest to the lowest numbered partition. This behavior is beneficial for performing reads on unique search key column values. If a matching index record was found, further lower numbered partitions do not have to be processed and the algorithm can break up earlier whereby read amplification is reduced. The returned candidate tuples are send to the visibility check in base table. The physical tuple reference indicates the entry point in version chain, which is processed until a tuple version is found that is visible to a transaction snapshot. In case of using an indirection layer, the entry point is provided by this structure. Order requirements for scans are processed afterwards. Read and scan operations can be accelerated by filter techniques. Bloom filters can reduce read amplification and latencies for point queries. In case of range queries, we decided to use a prefix bloom filter, however, this approach requires deep knowledge of the query set.

**Algorithm 4** Partitioned B-Tree - SCAN

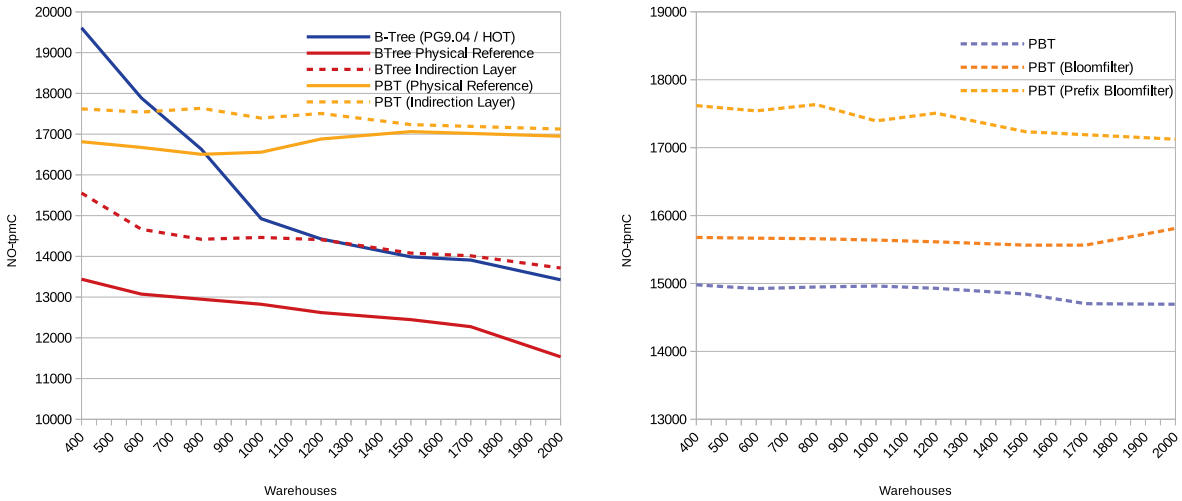
**Input:** Regular $|attr_{val,min}|, |attr_{val,max}|$
**Output:** $|references|$

```
1: procedure SCAN(|attr_val,min|, |attr_val,max|)                    ▷ scan w/ physical reference
2:     for each partition_scan ∈ PartitionsList                      ▷ begin with MAX(PartitionList)
3:     Let partitioned_record_min ← FORM_PARTITIONED_RECORD(partition_scan, |attr_val,min|)
4:     Let partitioned_record_max ← FORM_PARTITIONED_RECORD(partition_scan, |attr_val,max|)
5:
6:     if |attr_val,min|..|attr_val,max| ∈ partition_scan.filter then   ▷ if scan predicates allow utilization of filter
7:         Let ref ←FIND_IN(partitioned_record_min, partitioned_record_max)
8:         |references|.ADD(ref)
9:
10:        loop
11:
12:            if not HASNEXT( ) then
13:                break
14:
15:            end if
16:            Let ref ←NEXT( )
17:            |references|.ADD(ref)
18:
19:        end loop
20:    end if
21:    return |references|                                           ▷ send to visibility check in base table
22: end procedure
```

# 6 Experimental Evaluation

We present the analysis of Partitioned B-Trees (PBT) in comparison to traditional B$^+$-Trees in PostgreSQL 9.04; a relational DBMS with MVCC transaction management scheme. Typically, PostgreSQL uses an old-to-new (O2N) version ordering and physical tuple version storage. Index records have a physical reference to items located in base tables – denoted as B-Tree (PG9.04/HOT). PostgreSQL base table storage was modified to Snapshot Isolation Append Storage (SIAS) with a beneficial append-only write pattern and new-to-old (N2O) version ordering. We implemented and evaluated B$^+$-Trees and PBT with physical and logical tuple reference for SIAS.

We deployed PostgreSQL 9.04 and PostgreSQL with SIAS on an *Ubuntu 16.04.4 LTS* server with *Intel(R) Xeon(R) CPU E5-1620 v3 @ 3.50GHz* eight core processor, two 16GB *RIMM Synchronous 2133 MHz (0.5 ns)* RAM and an *Intel SSDPEDME400G4 SSD P3600 400GB, NVMe PCIe 3.0 x 4, MLC HHHL AIC 20nm 3DWPD* secondary storage drive. We used the well-known DBT-2 [1] TPC-C-like OLTP benchmark,




(a) Indirection Layer vs. Physical Tuple Reference

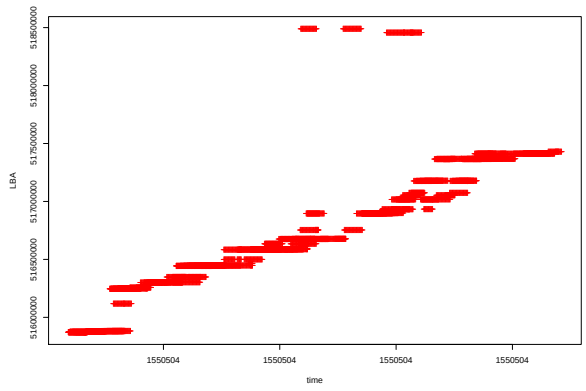
(b) Influence of Filter Techniques on Throughput

Figure 3: OLTP Benchmark Throughput Evaluation

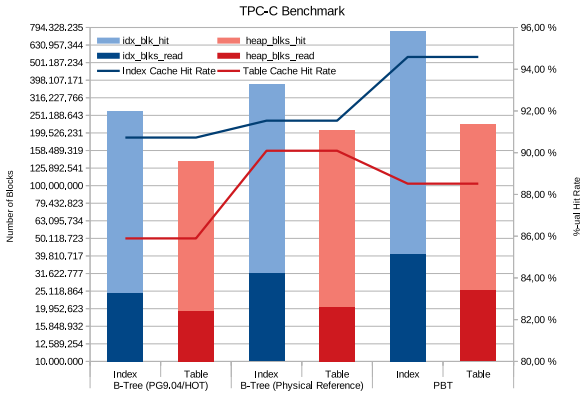
First, we evaluate throughput of B-Tree (PG9.04/HOT) as well as SIAS with B$^+$-Trees with physical reference and indirection layer in the DBT-2 benchmark. In Figure 3a, we show the throughput for different dataset sizes. The buffer cache of the DBMS is fixed to 600MB. The dataset size increase with the number of warehouses. B-Tree (PG9.04/HOT) performs well as long as most buffers for modifications are located in the buffer cache. Updates are performed in base tables by Heap-Only Tuples. The index maintenance effort is low due to this indirection on base tables – however, modifications to search key attributes have to be indexed. If the workload becomes write-intensive, the throughput falls rapidly. SIAS has a

scalable throughput (up to 30% improvement [3]), but increased effort in index management shrinks performance with physical reference B$^+$-Tree updates. With an indirection layer, index management is reduced to inserts of new tuples and changes in search key attributes. As a result, the throughput is increased by up to 20% and SIAS performs better than PostgreSQL at 1200 warehouses. Effects of indirection layer on index management are pretty low for PBT (see Figure 3a). The difference in absolute throughput is 6% at the dataset of 1000 warehouses. As the dataset grows and concurrent modifications become more uncommon, there is almost no difference in throughput between PBT with physical reference and indirection layer. The index is able to absorb additional modifications very well. PBT with SIAS has a 50% increased throughput in comparison to comparable B$_+$-Trees with physical references and about 30% with indirection layer at 2000 warehouses. The append-only approaches (SIAS and PBT) outperform PostgreSQL 9.04 with Heap-Only Tuples, as the benchmark becomes write-intensive at 700 warehouses. Their scalable throughput enable an improvement of 30% at 2000 warehouses.

Partitioned B-Trees (PBT) append modifications to the dataset in a mutable main memory partition. Effort of look-ups and especially of scans increase by number of partitions (see Figure 3b PBT with indirection layer), because in theory every partition has to be traversed. Up to 25 partitions were created for update-intensive indexes over the test duration. Point queries can break look-up on first matching record, which is visible to a transaction snapshot. Bloom filters are created at eviction time, if the partition becomes immutable and is written to secondary storage. Point queries can skip partitions, based on bloom filters and increase throughput up to 10%. The benchmark includes several scans. Prefix bloom filters include a fixed set of scan attributes and increase total throughput up to another 10%.




(a) Sequential Write Pattern of PBT at Partition Eviction Time




(b) Requests / Cache Hit Rate

Figure 4: Evaluation of Index Properties

We evaluated the write pattern of PBT (see Figure 4a). The diagram indicates the eviction of a main memory partition to secondary storage. The write I/O time is about 1ms. The logical

block addresses are provided by the file system. Each red cross indicates the write of a single index node. Several parallel and sequential writes of file extends can be identified in the diagram. Once an index node was written to secondary storage, its contents never change. PBT achieves the desired beneficial write pattern for semiconductor based secondary storage media.

In Figure 4b we show the requests on index nodes (blue) and base table nodes (red) for an write-heavy OLTP benchmark. Furthermore, the cache hit rate can be seen. Requests on cached nodes are displayed brighter than fetches from secondary storage. The scale of requests is logarithmic. The results are calculated for equal throughput over the test duration and all tables and indexes after ramp-up time. PBT requires more requests on index nodes due to partitioning of index records and bigger record sizes. Most requests are on buffered nodes, because many queries can be answered in the main memory partition. Index records of recent tuple versions are common to be located there. Requests can benefit from better cache hit rate in comparison to $B^+$-Trees. Immutable partitions on secondary storage are well protected by filters.

# 7 Conclusion

We presented different design decisions in MVCC transaction management scheme for large-scale data sets and update-intensive OLTP workloads, regarding the characteristics of modern storage technologies. We outlined resource-efficient append-only version-organization in base tables and its conflict with index management. We firstly implemented Partitioned B-Trees in a DBMS with MVCC transaction management scheme and evaluated their throughput and characteristics. PBT achieves an up to 50% increased throughput in relation to comparable $B^+$–Trees.

# 8 Acknowledgments

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program 'Services Computing'.

# References

[1] Database Test Suite - Browse /dbt2 at SourceForge.net, 2019. Accessed: 2019-03-01.

[2] R. Bayer and K. Unterauer. Prefix b-trees. *ACM Trans. Database Syst.*, 2(1):11–26, Mar. 1977.

[3] R. Gottstein. *Impact of new storage technologies on an OLTP DBMS, its architecture and algorithms.* PhD thesis, Technische Universität, Darmstadt, 2016.

[4] R. Gottstein, R. Goyal, S. Hardock, I. Petrov, and A. Buchmann. Mv-idx: Indexing in multi-version databases. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, IDEAS '14, pages 142–148, New York, NY, USA, 2014. ACM.

[5] G. Graefe. Sorting and indexing with partitioned b-trees. In *CIDR*, 2003.

[6] G. Graefe. Write-optimized b-trees. pages 672–683. VLDB Endowment, 2004.

[7] G. Graefe. Modern b-tree techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2011.

[8] P. L. Lehman and s. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, Dec. 1981.

[9] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[10] R. Sears and R. Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, 2012.

[11] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, Mar. 2017.