MV-PBT: Multi-Version Indexing for Large Datasets and HTAP Workloads

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

Robert Gottstein
Data Management Lab,
Reutlingen University, Germany
robert.gottstein@reutlingen-university.de

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

ABSTRACT

Modern mixed (HTAP) workloads execute fast update-transactions and long-running analytical queries on the same dataset and system. In multi-version (MVCC) systems, such workloads result in many short-lived versions and long version-chains as well as in increased and frequent maintenance overhead.

Consequently, the *index pressure* increases significantly. Firstly, the frequent modifications cause frequent creation of new versions, yielding a surge in index maintenance overhead. Secondly and more importantly, index-scans incur extra I/O overhead to determine, which of the resulting tuple-versions are visible to the executing transaction (visibility-check) as current designs only store version/timestamp information in the base table – not in the index. Such *index-only visibility-check* is critical for HTAP workloads on large datasets.

In this paper we propose the *Multi-Version Partitioned B-Tree (MV-PBT)* as a version-aware index structure, supporting index-only visibility checks and flash-friendly I/O patterns. The experimental evaluation indicates a 2x improvement for analytical queries and 15% higher transactional throughput under HTAP workloads. MV-PBT offers 40% higher tx. throughput compared to WiredTiger's LSM-Tree implementation under YCSB.

1 INTRODUCTION

The spread of large-scale, data-intensive, real-time analytical applications is increasing. Such applications result in Hybrid Transactional and Analytical Processing workloads (*HTAP*) combining long running analytical queries (OLAP) as well as frequent and low-latency update transactions (OLTP) on the same dataset and even on the same system [19].

Multi-versioning is at the core of many approaches and system designs suitable for HTAP. Under *Multi-Version Concurrency Control (MVCC)* reading transactions, executing long-running queries, do not block the frequent low-latency modifying transactions. Under such approaches multiple versions of each data item (i.e. tuple) may physically co-exist, whereas every transaction operates against a snapshot of the database comprising all versions it is allowed to see for consistent execution. Read operations simply operate on the latest committed version, visible to them and are therefore never blocked, yielding good read performance and concurrency. An update operation produces a new version of the updated data item and invalidates the predecessor version.
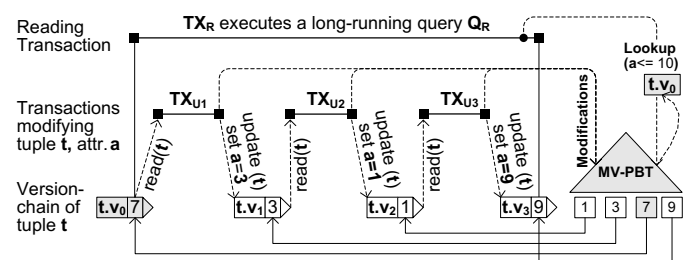
Figure 1: HTAP and Version-Chain Lengths: $TX_{U1} \ldots TX_{U3}$ **create new versions of tuple *t*, which are indexed. The index scan of** TX_R **returns only the index entries** ($t.v_0$) **visible to** TX_R **filtering the invisible ones** ($t.v_1 \ldots t.v_3$), **matching the search predicate.**

All versions of a tuple form a version-chain. Timestamps placed on every physical version-record are used to determine, which of the exisiting tuple-versions is *visible* to a transaction.

Under OLTP workloads, version-chains tend to be short, due to the predominantly short-lived transactions. For instance, under TPC-C the average version-chain length is approx. 1.2 [9]. *Under HTAP the DBMS needs to handle much longer version-chains due to the mix of long-running and short-lived transactions (Figure 1).* Whenever a transaction TX_R reads a tuple t the DBMS returns the latest version of that tuple $t.v_0$, committed before the start of TX_R. Even though, in the meantime multiple low-latency updating transactions $TX_{U1} \ldots TX_{U3}$ might have committed, producing successor-versions ($t.v_1 \ldots t.v_3$), $t.v_0$ cannot be garbage collected as long as, it is visible to an active transaction, i.e. TX_R. Thus, the amount of such transient versions can be as high as *several hundred millions* in real systems [14].

HTAP workloads in combination with long version-chains exercise significant pressure on indices. In a single-versioned system there is one index entry per tuple. However, in a multi-versioned system, the DBMS needs to index at least all committed tuple-versions (Figure 1), even the transient ones. Thus, long version-chains put extra pressure on the index. *Although most of today's systems are multi-versioned, the majority of index approaches still handle tuple-versions of the same tuple as if they were separate tuples, ignoring the version semantics.* If naïvely integrated, these slow down index lookups and may cause significant maintenance overhead to persistent indices, as index updates are very frequent and since index entries corresponding to obsolete tuple-versions need to be frequently garbage collected. Given the read/write

asymmetry of modern persistent storage technologies these operations result in prohibitively expensive in-place updates. In this context append-based index structures trading sequential writes for complex reads are a good candidate.

All in all, the following observations can be made:
1) *Version-obliviousness:* Although, all tuple-versions need to be indexed, current indexing approaches lack version information.
2) *Lack of index-only visibility-checks:* It is currently impossible to determine, which of the index-entries resulting from an index lookup/scan correspond to versions, visible to the calling transaction solely based on the index.
3) *I/O overhead:* Version-oblivious indices or naïve support for multi-versioning yield significant I/O overhead.

In the present paper we propose the *Multi-Version Partitioned B-Tree (MV-PBT)* as a version-aware index structure for MV-DBMS, in an attempt to address the above issues. MV-PBT is based on a variant of B^+-Trees called *Partitioned B-Trees* [13]. The contributions of this paper are:

- MV-PBT is a version-aware index structure. It contains version information and supports index-only visibility-checks.
- MV-PBT supports append-based write-behavior and exhibits much lower write-amplification compared to LSM-Trees.
- MV-PBT has been implemented in PostgreSQL. The performance evaluation under HTAP workloads (CH-Benchmark [2]) indicates 2x analytical throughput improvement due to index-only visibility-checks, while improving the transactional throughput by 15% compared to PostgreSQL's highly-optimized B^+-Tree. Under TPC-C MV-PBT performs 15% better.
- MV-PBT has also been implemented in WiredTiger (MongoDB). The performance evaluation indicates approx. 40% higher throughput under YSCB compared to WiredTiger's highly-optimized LSM-Trees.

The rest of the paper is organized as follows. We motivate the missing *version-awareness* and the need for *index-only visibility-checks* in Section 2, while Section 3 provides some background on various multi-versioning aspects. The design and implementation of MV-PBT is described in detail in Section 4, while the experimental evaluation is presented in Section 5. We. discuss related approaches in Section 6 and conclude in Section 7.

2 MOTIVATION

In this section we give a more comprehensive perspective on the above issues of: 1) *Version-obliviousness in indices*; 2) *missing index-only visibility-check*; and 3) *I/O overhead*. Consider the example in Figure 2, which is a more detailed version of Figure 1 with a conventional B^+-Tree. An initial transaction TX_{U0} (not depicted) inserts tuple t prior to TX_R, creating its initial version $t.v_0$. While TX_R is running, multiple concurrent transactions $TX_{U1} \ldots TX_{U3}$ update tuple t and each of them produces new versions of it ($t.v_1 \ldots t.v_3$). Only TX_{U3} inserts tuple y in its initial version $y.v_0$ in addition to creating $t.v_3$. Each tuple-version is a separate physical version record (Figure 2.A). It contains *version-information*: the *recordID* of the predecessor version and two timestamps, $t_{creation}$ - the timestamp of the transaction that created that tuple-version; and $t_{invladiation}$ the timestamp of the transaction that invalidated it by creating a successor version. The invalidation-timestamp is *null* if there is no successor. If a tuple gets deleted a special *tombstone* version-record is inserted to mark the logical end of the chain. *The version-information is only available on the version-record.*

Since version-records are independent physical entities they can be stored on any DB-page with enough free space. Figure 2.B depicts an example of the physical version-storage. *For consistency, an index on a table must contain index-entries for each committed version of every tuple.* Therefore, a B^+-Tree index *idx* on attribute *a* of table *R* (Figure 2.C) should reflect all versions of each tuple of *R*. *Since the index is version-oblivious it contains no version-information, and treats each tuple-version as if it were a separate tuple.* Consequently, if TX_R uses the index to count all tuples satisfying *"a ≤ 10"* (Figure 2.D), the index scan will return the matching index entries (referencing versions $t.v_0 \ldots t.v_3$). Now, each one of them must be checked for visibility, i.e. is it latest committed tuple-version prior to the start of TX_R. However, the necessary timestamps are available only on the version-records. Therefore, all of them are retrieved, at the cost of random I/Os.
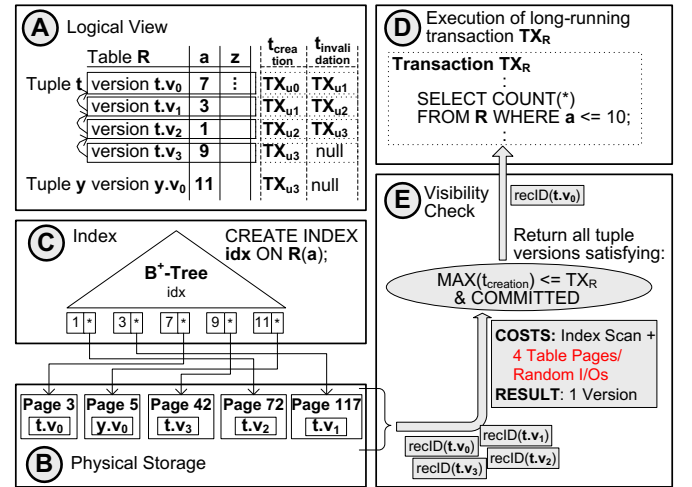
Figure 2: Index-Only Visibility-Check in Multi-Version DBMS: (a) logical tuples (t and y) of a table R and their versions; (b) the physical storage of these versions into database pages; (c) an index created over table R must index all versions; (d) an index-scan retrieves all versions matching the predicate, out of which (e) the visibility-check returns only the ones visible to calling transaction TX_R.

In our example (Figure 2.D, C and E), the index-scan for the condition *"a ≤ 10"* will return versions $t.v_3$, $t.v_0$, $t.v_1$ and $t.v_2$. Subsequently, they are read to extract the *version-information* ($t_{creation}$ and $t_{invalidation}$ – Figure 2.A) yielding four random I/Os. The visibility-check then determines the latest version committed prior to the start of TX_R, returning the recordID of $t.v_0$ and ignoring the rest. *Since the index is version-oblivious and thus does not support index-only visibility-checks, the I/O costs amount to: COST(Index-Scan) + 1 random I/O for each matching tuple-version. Especially for HTAP workloads this yields significant performance degradation depending on the length of the version-chains.*

To quantify the combined effect, we designed a simple experiment with *YCSB* [7] and PostgreSQL. We run YCSB workloads A (update) and E (scan) combined, performing frequent scans and updates. In parallel, we perform a point-query on a tuple every 30 seconds (simulating an HTAP workload). Additionally, we continuously increase the version-chain, by updating the tuple, until 50 versions are reached. In realistic HTAP settings, the amount of active versions can be as high as *several hundred millions*, while analyses can take as long as 1000s [14]. The experimental results
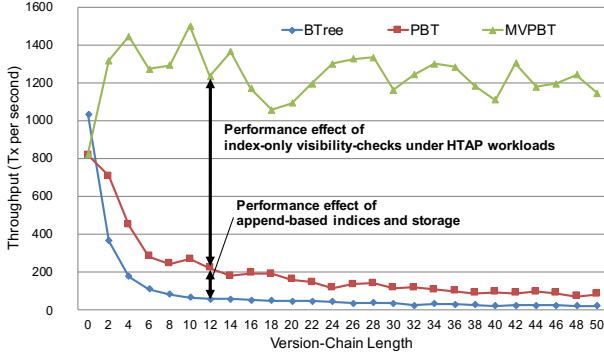
Figure 3: Performance Impact of Version Visibility Check.

are shown in Figure 3. The highly-optimized B$^+$-Tree implementation in PostgreSQL performs better than *MV-PBT* on a single tuple-version. However, as the version-chain length increases (6-8 versions) the performance drops rapidly to *approx. 50 transactions/sec*, due to version-obliviousness and random I/O. Basic Partitioned B-Trees (PBT), are likewise *version-oblivious*, but exhibit append-based write behaviour, avoiding in-place updates and perform therefore slightly better (*approx. 150 tx/sec*). Due to its *version-awareness* and support for *index-only visibility-check* MV-PBT exhibits much higher and robust performance (*approx. 1200 tx/sec*) with growing chain lengths. MV-PBT shows a performance increase with chain-lengths of two or more due to the partition buffer since: (a) the initial YCSB data load, producing the first version fills partition P_0 and evicts it; while (b) the second version is created by the benchmark workload and is in P_N.

3 BACKGROUND

Multi-Version Concurrency Control (MVCC) is one of the most popular transaction management schemes and is used in most modern DBMS: Oracle, Microsoft SQL Server, HyPer, SAP HANA, MongoDB WiredTiger, NuoDB, PostgreSQL or MySQL-InnoDB, just to name a few. These DBMS make different design decisions regarding various MVCC aspects described below.

3.1 Version Storage

Under MVCC a logical tuple corresponds to one or more tuple-versions (Figure 2.A). They form a singly linked list, which represents a version chain. There are two possible physical representations of a tuple-version (Figure 4): *physically materialized* or *delta-record based*. The former implies that each tuple-version record is stored physically materialized in its entirety and is in the focus of this paper. The latter implies that each modification of a logical tuple results in a delta-record, indicating the difference to another version (à la BW-Tree [15, 22]). The delta-records are connected and retrieved on demand by the DBMS storage manager to restore a tuple-version. *Delta-record based* system designs typically store a single version (oldest or newest) in the main store and use a separate store for the delta-records, which may be the undo log (à la InnoDB) or a temporary version store (à la MS SQL Server). Both organizations can perform modifications in-place or out-of-place. Out-of-place updates with *physically materialized* version-maintenance insert a new version-record in the base table. Based on the version ordering, additional modifications may be necessary to maintain logical timestamps or references.
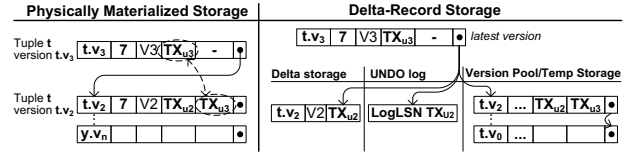

Figure 4: Version Storage Alternatives

Considering the characteristics of modern storage technologies, physically materialized version storage and out-of-place updates are preferable, due to lower write-amplification and the higher parallelism. Delta records tend to consume less space than materialized tuple-versions, but require additional processing and all predecessors or successors for tuple reconstruction.

3.2 Version Ordering

The set of tuple-versions of a database tuple is organized as a singly linked list. There are two different ordering methods (Figure 5): *old-to-new* and *new-to-old*.

Old-to-New ordering: The entry-point is the oldest tuple-version in version chain and each version contains a reference (recordID) to its successor. A visibility-check must therefore process all successors, beginning from the oldest tuple-version. This behavior is beneficial for lookups of long-running analytical (OLAP) queries under HTAP workloads, where older tuple-versions are likely to be the visible ones. Alternatively, OLTP workloads mostly require the newest version and would need to process the whole version chain. *New-to-Old ordering* implies that the entry-point is the newest tuple-version, which refers to its predecessor. Queries in the typically short OLTP transactions find the visible version very fast, but long-running OLAP queries may need to process several successors in version chain (Figure 3). *In-place* and *out-of-place* update strategies are are possible for both methods.

Considering the characteristics of modern storage technologies new-to-old ordering for physical version storage results in lower write-amplification and matches append-only storage. All other approaches require in-place updates.
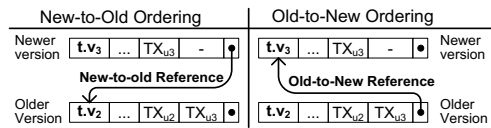

Figure 5: Version Ordering Alternatives

3.3 Version Invalidation Model

Under MVCC a version is said to be invalidated whenever a successor version exists. There are two possible invalidation models [9] (Figure 6). First, *two-point invalidation* is the state-of-the-art model, where the creation timestamp of the successor version is also placed as invalidation timestamp on the predecessor. *Two-point invalidation* works well with old-to-new ordering. However, with new-to-old ordering, the invalidation timestamp must be set on the predecessor version, yielding an in-place update and possibly a random write. Second, with *one-point invalidation* [11], the existence of a successor implicitly invalidates the predecessor and all version-records contain only the creation timestamp. *One-point invalidation* matches well new-to-old ordering, the use

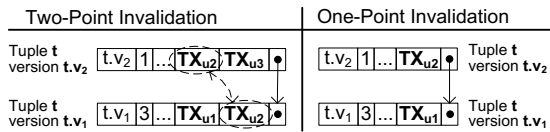of indirection layer (VIDs, and entry-points) as well as append-based storage.

Figure 6: Version Invalidation Model

3.4 Garbage Collection

Under MVCC modifications of a tuple result in the creation of a new tuple-version. Old tuple-versions become obsolete, if they are no longer visible to any of the active transactions. Therefore, some form of *version GC* is necessary to reclaim space and can improve performance. However, GC causes performance spikes (as it interferes with foreground I/O), reduces concurrency (as some form of locking is required) and increases write-amplification on secondary storage. GC [23] can be performed on transaction [14], tuple and index levels [15, 22]. Index-level GC (Section 4.6) purges index entries, resulting from index updates, maintenance or tuple-level GC.

3.5 Version/Index-Record Referencing

There are two possibilities to map index records to tuple-versions in base tables (Figure 7). *First*, classical physical references (recordIDs) can be used. Thus, the latest tuple-version in base tables (entry-point in the version chain) can be accessed directly, but changes to the latest version or its location result in index-record modifications. Such changes comprise: creation of a successor-version; storage management and physical movement (as in append storage) or garbage collection. *Second*, an indirection layer with logical references can be employed. Each tuple-version is augmented with an unique tuple-identifier (Virtual Tuple Identifier – VID), which is also stored in the index records. An index operation resolves the VID using a mapping table (indirection layer) to locate the physical entry-point. *An indirection layer can reduce index maintenance costs for in-place and out-of-place updates, but requires additional structures and processing.*
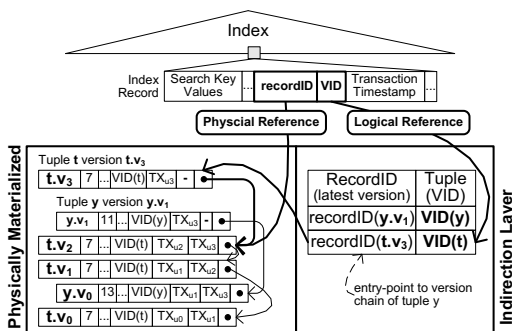

Figure 7: Version/Index-Record Referencing

Traditional index designs use physical references and contain no version-information, which tends to increase index maintenance overhead as well as the visibility check costs for lookups and scans. Alternatively, modern index-structures (BW-Tree) use an indirection layer, but contain no version-information and support no index-only visibility check. This can cause massive read amplification for mixed workloads. An optimal index structure should reduce write amplification and return only references to tuple-versions that are visible to a transaction snapshot. MV-PBT uses physical or logical references, is version-aware and produces append-only sequential write pattern.

3.6 Discussion

We have outlined some relevant design decisions for storing tuple-versions in multi-version DBMS. Modifications are preferably stored as *physically materialised tuple-versions* in base tables, rather than deltas, due to tuple reconstruction costs. Moreover, this enables direct access to each tuple-version from additional access paths. *Out-of-place updates* reduce write amplification to secondary storage. Garbage collection is required for space reclamation, but brings additional complexity to data structures.

A *new-to-old* version ordering requires index maintenance for every new tuple-version, because the entry-point of the version chain for that tuple changes. A logical indirection layer ensures fast lookups by efficiently returning the entry-point of a version chain and reduces index maintenance effort. *New-to-old* ordering is beneficial for OLTP and speeds up visibility-check as the latest version ist typically the visible one, yet older versions may require slow reconstruction. Alternatively, *old-to-new* ordering is supports long-running OLAP operations and visibility-check in HTAP settings, as the oldest version is directly accessible. Yet, modifications and maintenance may suffer low performance.

Indices for mixed workloads and large datasets should rather return visible tuple-versions. Alternatively, traditional index structures only return version candidates, which have to be subsequently verified, fetching version-records from base tables by performing random I/O. For these reasons *MV-PBT* rely on physically materialised versions, out-of-place updates, a new-to-old ordering, one-point invalidation and can do without an indirection layer.

3.7 Storage Characteristics

Modern database storage management needs to address the characteristics of semiconductor storage technologies [20]. Consider Figure 8, which deptics the I/O characteristics of the enterprise Flash storage used in the evaluation. Typical index search operations result in large amount of small (8K) random reads. Hence, optimize for read IOPS and sequential writes (≥64K). We derive the following tradeoffs for the I/O behaviour of MV-PBT: (a) transform random writes in sequential writes with higher granularity (MB); and (b) trade sequential writes for complex and possibly random reads with higher parallelism and smaller granularity (KB). Thus, append-based storage managers are beneficial for the base tables [9, 11]. Write-sequentialization is therefore necessary for indices, and MV-PBT supports it intrinsically, like LSM-Trees.

		Read		Write	
Blocksize [KB]		8	64	8	64
Sequential	Iops	122382	24180	11104	1343
	MB/s	956	1511	87	84
Random	Iops	112479	23631	7185	56
	MB/s	879	1477	1184	74

Figure 8: I/O Characteristics of Intel DC P3600 SSD.

4 MULTI-VERSION PARTITIONED B-TREES

Multi-Version Partitioned B-Trees (Figure 9) are based on Partitioned B-Trees (PBT), introduced by Goetz Graefe [12, 13]. PBT in turn represent an enhancement on traditional B$^+$-Trees[4]. PBT (and MV-PBT) create index partitions based on an artificial, leading key-column – *the partition number*. All index-entires in a partition have the same partition number in the search key. PBT (and MV-PBT) utilize a portion of the database buffer (*partition buffer*) to host the latest partition P_N, where insertions and updates to existing partitions ($P_0 \dots P_{N-1}$) are placed. Updates to existing index entries are treated as *replacement records* to avoid in-place updates. Once P_N gets full it is appended to persistent storage and becomes immutable.
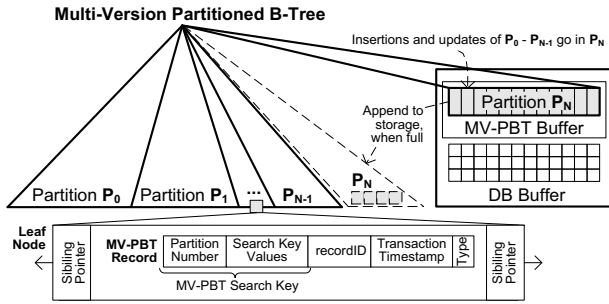
Figure 9: Structure of a Multi-Version Partitioned B-Tree.

Regular MV-PBT records comprize of a *partition number*, its *search key columns*, and a *recordID (set)*. Furthermore, MV-PBT index records contain version-information: *logical transaction timestamp* for validation or invalidation of the tuple-version and optionally an *unique virtual identifier* (indirection layer). Each partition number identifies a single partition. Partition numbers are unique, monotonically increasing, two-byte integer values. This enables the MV-PBT to maintain partitions within one single tree structure in alphanumeric sort order. The partition number is an artificial column and is therefore transparent to higher database layers. Each MV-PBT maintains partitions independent of other MV-PBTs. Partitions appear and vanish as simple as inserting or deleting records. They can be reorganized and optimized on-line in system-transaction merge steps, depending on the workload. Partitions can support additional functionalities, like bulk loads or can serve as multi-version store[13].

MV-PBTs write any modification of index records exactly once – upon eviction of a partition, except for later reorganization or garbage collection operations. This is realized by forcing sequential writes of all leaf nodes in a partition (Figure 9). Leaf nodes of modifiable main memory partitions are stored in a separate buffer cache – the *MV-PBT Buffer*. This area is shared for all MV-PBT indices in the database. Once the MV-PBT Buffer gets full, a victim MV-PBT is selected and its P_N is written to secondary storage. The MV-PBT Buffer is managed by a special replacement policy, giving active partitions the chance to grow (Section 4.5).

4.1 MV-PBT Record Types

Persistent index partitions are immutable. Direct modification-operations are forbidden. Therefore, modifications to existing index-records as well as insertions are placed in the buffered partition P_N. To handle this behavior MV-PBT introduces new index-record types. Currently the following are defined.
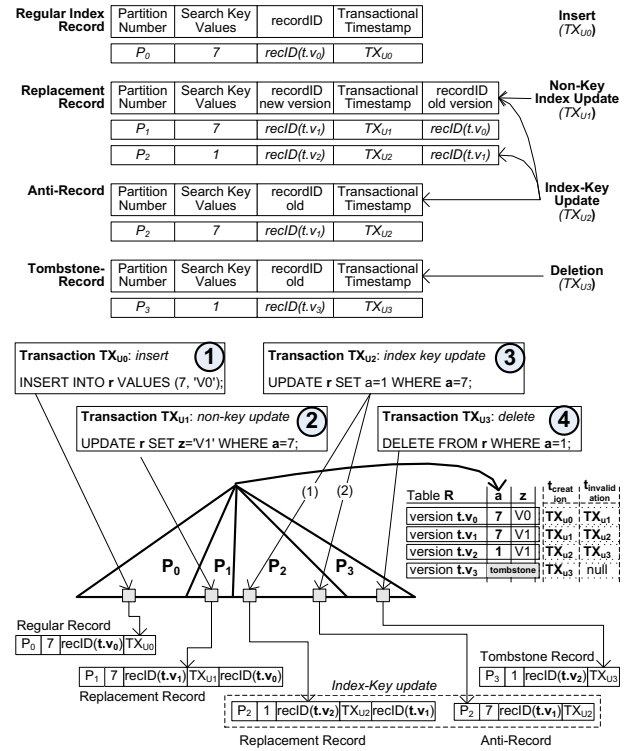
Figure 10: MV-PBT Index-Record Types and Their Use: MV-PBT record format (top), an example including a sequence of transactions and their index records (bottom).

Regular Index Records are created upon the insertion of new tuples. The *partition number* of the newest MV-PBT partition P_N is inserted together with the search key values. The *recordID* (pageID and slot) of the newly inserted tuple-version is included as well as the *transaction timestamp* of the inserting transaction (Figure 10). The latter is essential for index-only visibility-checks. For example, transaction TX_{U0} (Figure 10) inserts a new tuple (t), in its initial version ($t.v_0$), causing the creation of a *regular index record* in partition P_0.

Replacement-Records result from tuple-updates on *non-index key columns* on existing index-entries. Such updates yield a new tuple-version that becomes the new chain entry-point, which needs to be reflected in the index. Although the index-record for the previous version has not changed (non-index-key update) the version-information and recordID of the new version need to be replaced. However, this is not possible, if the index-record is already in an immutable partition ($P_0 \dots P_{N-1}$). Therefore a *replacement record* is inserted in the newest partition P_N to logically replace the old one with the recordID and the version-information. The *Replacement Record* (Figure 10) contains: the recordID of the new version, its creation-timestamp as well as the recordID of the predecessor version. Hence the record includes some "anti-matter" [13] (recordID) invalidating the predecessors as well as some "matter", i.e. recordID and timestamp if the new version. For example, transaction TX_{U1} (Figure 10) updates the attribute z of the previously inserted tuple (t), producing a new version ($t.v_1$). Although the index-key 7 remains unchanged, the version-information of ($t.v_1$) has to be updated, causing the creation of a *replacement-record* in partition P_1.

Anti-Records are required for updates on index-key attributes and are always used in combination with *replacement records* in the same partition. If the index-key of an existing index-record (in the immutable partitions) gets updated, MV-PBT inserts a combination of an *anti-record* and a *replacement record*. *Anti-records* are pure "anti-matter" as they mark the extinction of the old index record (from partitions $P_0 \ldots P_{N-1}$), whereas the simultaneously inserted *replacement record* represents the new "matter" and reflects the new index-key and the new version-information. The anti-record and the replacement record are inserted in P_N and are placed according to the sort-order of the search-key value. An anti-record contains the recordID of the old version, together with its search key and the transaction timestamp of the updating transaction (Figure 10). For example, transaction TX_{U2} (Figure 10) updates the indexed attribute *a* of the previously inserted and updated tuple (*t*), producing new version ($t.v_2$), modifying the index-key from *7* to *1*. The *anti-record* (marking the extinction of the replacement-record from partition P_1) reflects the recordID of the predecessor version ($t.v_1$), contains its index-key values (7) and the transaction-timestamp of the current updating-transaction (TX_{U2}). The simultaneously inserted replacement record reflects the new and updated value of the search key (i.e. *1*), the recordIDs of the old and the new tuple-versions ($t.v_1$ *and* $t.v_2$) as well as the transaction-timestamp of TX_{U2}. Since the index records are kept in sort order of the search-key values within a partition (as in a B-Tree), the replacement record is placed first in order, followed by the anti-record.

Tombstone-records indicate the deletion of a tuple. If a tuple is logically deleted, it does not become erased immediately in MV-DBMS, because it could be visible to a concurrent transaction. Rather a tombstone tuple-version record is inserted in the DB, which needs to be reflected in the MV-PBT index. *Tombstone-records* are similar to *Anti Records* in that they represent pure "anti-matter", marking the extinction of the whole tuple-version chain. The difference is that if a tombstone-record is visible to a transaction, no further tuple-version belonging to this chain can be visible, even no *replacement record*. *Tombstone-records* (Figure 10) contain the recordID of the latest tuple-version and the transaction-timestamp of the deleting transaction.

For example, transaction TX_{U3} (Figure 10) deletes tuple (*t*), creating a tombstone-version ($t.v_3$) in the DB. Therefore a tombstone record is inserted in partition P_3 with the recordID of the deleted tuple-version $t.v_2$, reflecting deletion of the whole version chain $t.v_2 \rightarrow t.v_1 \rightarrow t.v_0$.

4.2 MV-PBT Operations

In the following we describe the index operations in an MV-PBT:

- *Insert Operations* are only performed in P_N. An *insertion* yields the creation of an *regular index-record* in P_N with the recordID of the newly created tuple-version and the timestamp of the creating transaction. The insertion traverses the buffered partition P_N and places the new index record according to the alphanumeric sort-order of the search-key (ordering issues are described in Section 4.3). The MV-PBT buffer management strategy (Section 4.5) guarantees sufficient space for the insertion and possible maintenance. In case of an *non-unique index* the insertion is performed directly. Alternatively, given a *unique index*, a lookup operation (see Search and Scan) is performed ahead of the inseartion to guarantee the non-existence of the new index-key.

- *Update Operations* are performed in different ways. If a transaction modifies a tuple-version in a way that a non-index-key attribute is changed (*non-key update*) a new tuple-version is created and its version-information needs to be reflected in the index. In case of *non-key updates* MV-PBT inserts a *replacement-record* in P_N (Figure 10), containing the version-information (recordID and timestamp) of the modifying transaction. By doing so, it logically *replaces* the index-record, which is located in an older partition, and reflects the predecessor version. Alternatively, if the modifying transaction updates an index-key attribute (*index-key update*) a *replacement record* as well as an *anti-record* are inserted in P_N (ordering issues are described in Section 4.3). The former reflects the new and modified index-key value in the new tuple-version, the latter indicates the extinction of the old index-record, reflecting the index-key value of the predecessor version. In case of an *unique index*, the MV-PBT first performs a lookup to ensure the non-existence of the new key-value.

- *Delete Operations* cause the insertion of a *tombstone record* in P_N. If a transaction deletes a logical-tuple a tombstone version is created indicating the deletion of the whole version-chain, to transactions to which it is visible. Analogously, MV-PBT inserts a *tombstone record* to indicate the extinction of all index-records corresponding to the version chain. Ordering issues are described in Section 4.3.

- *Search and Scan Operations* process partitions in reverse order from P_N to P_0. Filter techniques such as *Partition Range Keys*, *Minimum Transaction Timestamp* or *Bloom- and Range Filters* (Section 4.7) are needed for selecting the predeceasing partition which may contain an index record, matching the search conditions (Algorithm 1). The search conditions are extended to match the format of an MV-PBT – the partition number is prepended to the first search key column. A regular root-to-leaf traversal operation is performed and the cursor is positioned. Afterwards, the next matching index record is requested and checked for visibility (Section 4.4). This process is repeated until an index record, visible to current transaction is found, and can be returned together with the respective *recordID*. *Partition number* and *timestamp* are transparent for higher database layers. Index records of most recent tuple versions are found and processed first, due to index-record ordering (Section 4.3), which is very beneficial for simple search conditions, like point lookups.

  Complex scan operations (Algorithm 2) build a set of all matching index records, spreading all MV-PBT partitions. Every partition is pre-selected by filter techniques and processed from P_N to P_0. Traversal operations benefit from commonly buffered higher levels of the tree-structure. Matching index records of any record type in a partition are processed by the index-only visibility-check. Visible index records are added to the result set without *partition number* and *timestamp* in regular sort order. If no further index record matches the scan conditions, the algorithm proceeds with the preceding partition. Finally, the result set is returned. It is filled with all index records (including *recordIDs*), matching the scan and visibility conditions of the calling transaction.

  A single scan process without rechecking for concurrent modifications in P_N is sufficient, due to transaction snapshots as concurrent modifications in P_N are invisible, anyway. Expensive retrieval of version-records from the base-table (*random read I/O*) for *version-information* is avoided. In case of selection of non-index attributes, the recordID indicates the location

of version-record in the base-table, which can be directly accessed.

Algorithm 1 MV-PBT Search

1: **function** SEARCH(Search conditions $|attr_{val,cond}|, ...$)
2: **Output:** *IndexRecord*
3:    **while** HASNEXT( ) **do**
4:       Let $idx\_record \leftarrow$ NEXT( ) ▷ fetch next index record
5:       **if** VISCHECK(idx_record) **equals** *VISIBLE* **then**
6:          **return**   SET_RETURN_FORMAT(idx_record)
                  ▷ hide *partitionnumber* and *timestamp*
7:    **while** $part \leftarrow$ PREVIOUSPARTITION($part$) **do**
8:       **if** $|attr_{val,cond}| \in part.filter$ **then**
9:          Let $|skeys_{part}|$ $\leftarrow$
               FORM_REC($part, |attr_{val,cond}|$)
10:          TRAVERSE($|skeys_{part}|$)
11:          **return** SEARCH( )
12:    **return** $\emptyset$

Algorithm 2 MV-PBT Scan

1: **function** SCAN(Scan conditions $|attr_{val,cond}|, ...$)
2: **Output:** ResultSet of $|IndexRecords|$
3:    $part \leftarrow \emptyset$ ▷ PREVIOUSPARTITION returns P_N for $\emptyset$
4:    **while** $part \leftarrow$ PREVIOUSPARTITION($part$) **do**
5:       **if** $|attr_{val,cond}| \in part.filter$ **then**
6:          Let $|skeys_{part}|$ $\leftarrow$
               FORM_REC($part, |attr_{val,cond}|$)
7:          TRAVERSE($|skeys_{part}|$)
8:       **while** HASNEXT( ) **do**
9:          Let $idx\_record \leftarrow$ NEXT( ) ▷ neighbor in BTree
10:          **if** VISCHECK(idx_record) **equals** *VISIBLE* **then**
11:             $|IndexRecords|$.ADD(
               SET_RETURN_FORMAT(idx_record))
12:    **return** $|IndexRecords|$

4.3 MV-PBT Index-Record(Version) Ordering

The version/partition-placement in MV-PBT is governed by modification, search and scan algorithms. *Index-records of predecessor versions are likely to be located in lower-numbered partitions, successors in higher-numbered ones (Figure 10)*. This however necessitates multiple memory partitions for an MV-PBT.

To address such issues the current MV-PBT design uses a single main-memory partition P_N for each MV-PBT. *However, for index-records with the same index-key it is mandatory that records for newer/successor versions are always placed before index-records for older/predecessor versions in P_N.* In other words the *primary sort-order* of the index-records in a P_N is on the search-key (mostly descending), however all records with the same search-key are sorted in inverse *secondary sort-order* (mostly ascending) on the transactional timestamp.

Search and scan operations traverse partitions *backwards*: starting from buffered partition P_N (i.e. $P_N \rightarrow P_{N-1} \cdots \rightarrow P_0$). Yet, given the above ordering, index-records of newer tuple-versions, matching the search predicates, are processed *first* in forward direction (i.e. in descending timestamp-order). Only then the next lower-numbered partition is traversed and processed. *This is how MV-PBT ensures that in a search and scan operation,*

newer versions can always be found before older ones in the same partition, and across partitions.

Consider for example Figure 11, where we have only two partitions and index-records reflecting updates to the same tuple go to P_1, and contrast to Figure 10, where all index-records with higher-timestamps are placed in higher-numbered partitions. Observe that the index-records in P_1 (Figure 11) appear in their primary-order (on the search key), i.e. records with search-key 1 precede those with 7. Observe also that the *tombstone record* with key 1 precedes the *regular record* as a result of the *secondary sort-order* since $timestamp(TX_{U3}) > timestamp(TX_{U2})$.
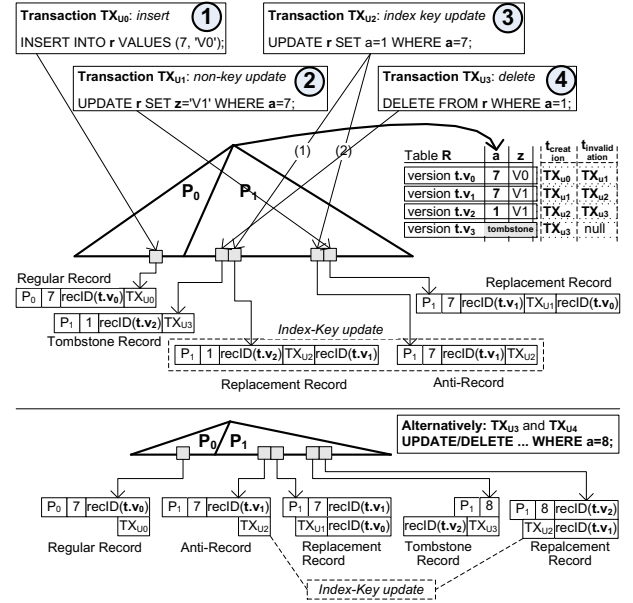

Figure 11: MV-PBT Index-Record Ordering.

4.4 MV-PBT Index-Only Visibility-Check

MV-PBT is *version-aware* and supports *index-only visibility-check*, i.e. it returns a set of index records matching the search condition and *visible* to the calling transaction. In doing so, MV-PBT avoids the expensive retrieval of base-table version-records to extract their *version-information*.

The *index-only visibility-check* (Algorithm 3) is inherently supported by the data structure. MV-PBT index records (Section 4.1) contain version-information and define modifications and recordIDs of tuple-versions. The respective *index-record ordering* is essential to scans (Section 4.3), whereby records indicating the invalidation of a tuple-version are guaranteed to be placed before the *"validating"-records* for a given transactional timestamp.

Index records of any type, matching the search-conditions are processed by the visibility check. They are *invisible* to a transaction, if:
(a) the index record is *flagged for garbage collection*;
(b) the transaction timestamp of the index-record is *greater than* the timestamp of the calling transaction; or
the transaction corresponding to the index-record timestamp is *concurrent* to the calling transaction;
(c) visible record with *anti-matter* for the recordID (anti-matter, replacement- and tombstone-records) was already encountered (in this case also checked for GC); or

(d) the index record is either a *tombstone record* or an *anti-record*. An additional visibility-check by processing the version chain in base table is not required. Skewed updates on tuples do not lower the performance of the index-only visibility check, due to well performing garbage collection and well-cached version-chains in the main-memory partition P_N.

Algorithm 3 MV-PBT Index-Only Visibility-Check

1: **function** VISIBILITYCHECK(*idx_record*)
2: **input:** *idx_record* at current scan position
3: **output:** *BOOL_VISIBLE*
4:    Let *anti_map* ← *Map* of (*recID*|*TS*)   ▷ anti-matter
5:    **if** IS_SET(*idx_record*, *FLAG_GC*) **then**
6:        **return** *INVISIBLE*
7:    **if not** PRECEDES(*idx_record.ts*, *CurrentTxId*) **OR** ISCONCURRENT(*idx_record.ts*, *CurrentTxId*) **then**
8:        **return** *INVISIBLE*
9:    **if** ts_{anti} ← *anti_map*.GET(*idx_record.recID_{matter}*) **and** PRECEDES(*idx_record.ts*, ts_{anti}) **then**
10:        CHECKFORGC(*idx_record*)
11:        **return** *INVISIBLE*
12:    **if** IS_SET(*idx_record*, *FLAG_ANTI_MATTER*) **then**
13:        *anti_map*.PUT(*idx_record.recID_{anti}*, *idx_record.ts*)
14:    **if** IS_SET(*idx_record*, *FLAG_MATTER*) **then**
15:        **return** *VISIBLE*
16:    **return** *INVISIBLE*

4.5 MV-PBT Buffer Management

MV-PBT accumulate modifications to persistent partitions in the latest partition P_N, which is held in the MV-PBT partition buffer (Figure 9). All MV-PBT indices place their respective P_N in the MV-PBT buffer, which rises the question of the proper buffer management strategy. Well-known replacement policies (like LRU or ARC) are not suitable for managing the set of leaf nodes contained in the respective P_N as well as different P_N. The MV-PBT buffer should (a) only evict partitions as a whole instead of individual pages (like in LRU) to achieve sequential write patterns; and (b) give partitions of update intensive indices a fair chance to grow, and balance it across all indices. Remember that MV-PBT read operations place persistent partition nodes in the main/shared DB-Buffer. MV-PBT buffer-management strategy can be summarized as follows. Whenever the buffer-size threshold is reached the MV-PBT buffer manager selects the largest partition of all indices as a *victim* for eviction. Smaller, less update-intensive partitions are frequently evicted to avoid imbalanced number of partitions per MV-PBT and shrinking partition sizes.

The *eviction process* (Algorithm 4) can be summarized as follows. A new partition numbered P_{N+1} (initially P_{N+2}) is created for ongoing modifications. The current victim partition P_N becomes *immutable* and is scanned, as following operations are performed cooperatively and latch-free, piggybacking that *scan*.

(1) Version-chains are built for each of the Scan-ResultSet records, using their timestamps and RecordIDs, and creating a temporary VID for each chain. While doing that, obsolete index-records (parts of the version-chain) are detected and marked for garbage collection.
(2) *Garbage Collection* is performed on the marked records (no longer needed/invisible records are removed) and the result is *written out* to new leaf nodes.

(3) During this process index-records and leaf nodes are transformed to an on-disk format, whereby prefix-truncation, compression and encoding as well as dense-packing (Section 4.7) are performed. Furthermore, the partition number of each index record is decremented from P_N to P_{N-1}. Now P_{N-1} is a separate partition, which is yet unknown in the MV-PBT partition metadata. The process resembles a leaf-build in PostgreSQL: having full leaf pages the intermediary index nodes can be easily built on top. Concurrent, lookups and scans are still performed on the old P_N nodes.
(4) In parallel, well-sized *(prefix-) bloom filters* are created (Section 4.7).
(5) Dense-packing, compression and read-optimizations are performed to higher level intermediary nodes, resembling a bottom-up build. All nodes are *sequentially written out*.
(6) Finally, P_{N-1} is added to the MV-PBT partition metadata. The old P_N leaf nodes, on which concurrent non-blocking reads had been executing, are detached from the MV-PBT and are freed for reuse.

Algorithm 4 MV-PBT Partition Eviction

1: **function** EVICT($|P_N|$)
2: **Input:** set of P_N in MV-PBT buffer
3:    Let p_{evict} ←SELECTEVICTIONVICTIM($|P_N|$)
4:    Add *Partition* $p_{evict+1}$ to B$^+$-Tree *PartitionsList*
5:    SET(p_{evict}, *FLAG_IMMUTABLE*)
6:    Let *recordSet* ← SCANRECORDS(p_{evict})
7:    GARBAGECOLLECTIONP3(*recordSet*)
8:    *worker*1.LOADANDFLUSH($p_{evict}.pNo - 1$, *recordSet*)
9:    *worker*2.CREATEFILTERS(p_{evict}, *recordSet*)
10:    WAIT( )
11:    Let p_{evict_new} ←DECREMENTPARTITIONNUMBER(p_{evict})
12:    DETATCHANDFREE(p_{evict})

4.6 MV-PBT Partition Garbage Collection

Mixed workloads with high update-rates result in massive amount of tuple-versions, which need to be garbage collected once a long-running reading/analytical query completes [14]. Same is true for the corresponding index-records. With high probability these records are located in the main-memory partition P_N of an MV-PBT due to their temporal locality. Therefore, we implemented a cooperative page-level garbage collection (GC) for P_N.

Phase (1): The GC *piggybacks* regular index-scans to identify index-records of versions, that are not visible to any active transaction (*cutoff-transaction*). As a page is already latched (shared), the following checks a performed on each record: (a) comparison with the lowest active transaction timestamp and if lower, mark predecessors as *victim-versions* for GC; (b) if higher, but a successor exists, mark all predecessors as victims for GC. In both cases, a *hasGarbage* flag is set in the page header (no exclusive latch required). This step also *piggybacks* the in-memory structures of the scan and index-only visibility check algorithms. Records with *anti-matter* (anti-matter, replacement and tombstone records) require special attention, as they are still required for invalidation of predecessors. Hence the *anti-matter* record with the highest timestamp smaller than *cutoff* transaction timestamp must not be garbage collected. Index-record ordering (Section 4.3) supports GC while scanning, since successors are mostly processed first. *Phase (2):* Update operations check the *hasGarbage* flag in page header. If set they first set the *recordID* of the oldest required

record with *anti-matter* (anti-matter, replacement and tombstone records) to the recordID of the oldest victim-version of that chain on the page. Next, GC victims are removed on that page, the space is reclaimed and only then the update operation proceeds. This behavior saves memory, speeds up scans and visibility checks as well as reduces index maintenance operations (split).

Phase (3): To handle version-chains spanning several pages, and for final cleanup before partition eviction the whole partition is scanned and the version chains (based on timestamps and records) are built in memory. This scan is also piggybacked for filter creation and dense-packing (Section 4.7). Before switching to sibling page, obsolete versions are removed after updating invalidation reference and in-memory version chain is updated.

4.7 MV-PBT Filters and Optimizations

Various optimizations can be performed, based on the fact that once written to storage MV-PBT partitions are *immutable*.

Bloom Filters. Each MV-PBT partition has a bloom filter (*BF*) on the search key. Using bloom filters accelerates key lookups (point-queries) in a partition, by avoiding unnecessary scans. Whenever a key lookup is performed, a BF-query executed first, to verify whether the key does *not* exist in the partition. If it does not exist MV-PBT proceeds with the next partition. Alternatively, if the BF returns true (i.e. the key may exist), MV-PBT scans the whole partition.

Our experimental evaluation (Figure 13) indicates that the average BF size is small – in the order of few hundred KB. Therefore frequently used filters are usually cached in the MV-PBT buffer. Furthermore, their precision is *98%* on average, thus false positives and therefore superfluous scans are rare. BF is is computed efficiently on eviction, piggybacking existing maintenance scan and is persisted as part of the partition metadata.

Range Filters. Partition bloom filters accelerate point lookups, but cannot handle range predicates. Currently, we employ *prefix Bloom Filters (pBF)*, if appropriate, to speedup range scans.

Dense-packed, Read-Optimized immutable storage. Since a partition is immutable once persisted, various space and read-optimization techniques can be applied. *Dense-packing* is used to perform coalescing and free-space optimzation. When in-memory leaf nodes are on average 67% full to accumulate modifications and avoid splitting, however when persisted the the space utilization can be maximized. MV-PBT performs *dense-packing* as part of the final garbage collection and space reclamation.

Especially for *non-unique indices* MV-PBT performs *reconceliation* upon eviction to convert all regular records with the same search key to a single regular record with a set of {recordID, timestamp}, instead of holding separate record for each key instance. The same is true for replacement records, where for the same search key sets of {$recordID_{NEW}$, $Timestamp_{NEW}$, $recordID_{OLD}$} are created. Last but not least, compression techniques such as *prefix-truncation* or *delta-compression* are performed on the search key. Along the same lines, various read and cache-aware optimizations can be performed.

5 EXPERIMENTAL EVALUATION

We present the analysis of Partitioned B-Trees (PBT) and MV-PBT together with traditional B$^+$-Trees (which serve as *baseline*) in PostgreSQL 9.04. Standard, PostgreSQL uses an *old-to-new* version ordering, *physically materialized version storage* and *two-point invalidation*. Index records have a physical reference to base tables – denoted as B-Tree (PG/HOT). PostgreSQL base

table storage was also modified to Snapshot Isolation Append Storage (SIAS) [9, 11] with a beneficial append-only write pattern, one-point invalidation and new-to-old version ordering. We implemented and evaluated B$^+$-Trees and PBT with physical references and with logical tuple references on top of SIAS [9, 11].

Experimental Setup. We deployed PostgreSQL 9.04 and PostgreSQL with SIAS [11] on an *Ubuntu 16.04.4 LTS* server with an eight core *Intel(R) Xeon(R) E5-1620* CPU, 2GB RAM and an *Intel DC P3600 400GB SSD* drive. We used the well-known DBT-2[1] TPC-C-like OLTP benchmark and mixed workload CH-Benchmark [6] in OLTP-Bench [2, 8] for experimental evaluation. The OS page cache is cleaned every second to ensure repeatable and reliable results (even though conservative).

Mixed Workloads: CH-Benchmark. MV-PBT is designed for large datasets and mixed workloads. We evaluate the throughput of B$^+$-Trees, PBT and MV-PBT under the CH-Benchmark [6] in OLTP-Bench [2, 8]. MV-PBT *doubles* the analytical throughput compared to B$^+$-Trees (Figure 12a), improving it from 0.29 to 0.61 queries/transactions per minute. In the same time, MV-PBT yield *15%* higher transactional throughput than B$^+$-Trees (Figure 12a).

The performance improvements are mainly due to *index-only visibility-check* and *partition garbage collection*. To illustrate the combined effect we turn off both and repeat the experiment. Consider now the lower MV-PBT performance bars in Figure 12a. Without *partition garbage collection* and *index-only visibility-check* the OLAP performance drops by 75% from 0.61 to 0.16 queries per minute, whereas the OLTP throughput plummets from 4232 from to 3093 tx/min.

Mixed Workloads: Index-Only Visibility-Check and Garbage Collection. In a further experiment we investigate MV-PBT GC and visibility-check in more detail varying the version-chain length. We run the OLTP part of the CH-Benchmark and execute a query on the same dataset (Figure 12b), however we pause it (using *pg_sleep*) for 30/60/90/120 seconds to simulate a long-running query and vary the amount of transient versions and the chain length. Clearly, as the version-chain length increases, index-only visibility-checks gain importance, because unnecessary read I/O on base table can be reduced.

We compare PBT and standard *visibility-check in base table (VC)* to MV-PBT and *index-only visibility-check (idxVC)* (Figure 12b). As the query processing time and version-chain length increase, index scans and *VC* of slow down PBT by an order of magnitude. Even if the version-chain length has no linear growth, pages in base table get evicted and need to be fetched more frequently. MV-PBT performs *idxVC* however without garbage collection (Figure 12b *MV-PBT w/o GC*), every index record of successor tuple-versions has to be processed, likewise the scan time increases proportionally with the length of the version-chain. With garbage collection (Figure 12b *MV-PBT w/ GC*), the number of scanned index records and the scan time remain almost constant. However, GC requires additional processing and latches index nodes in P_N. Reading transactions have to wait for latches and scan time increases - consider Figure 12b at 30 seconds sleep time. As more index record get garbage collected, GC improves the index scan time - compare MV-PBT with and without GC at 30 and 120 second (Figure 12b).

Sequential write-pattern/Append-based storage. Based on the tradeoffs derived in Section 3.7. MV-PBT needs to support write sequentialization and append based storage. In this experiment we evaluate the write pattern of MV-PBT (Figure 12c). Using *blktrace* and *blkparse* we record an I/O trace during the partition eviction from MV-PBT buffer. The X-axis represents the
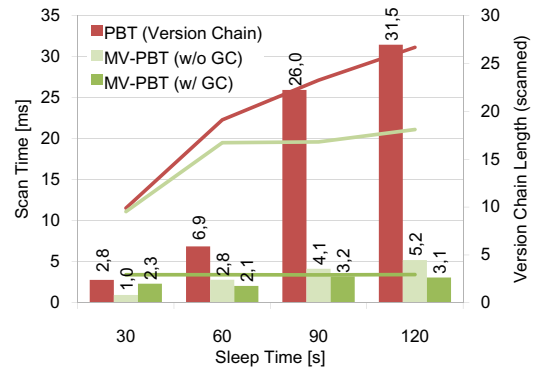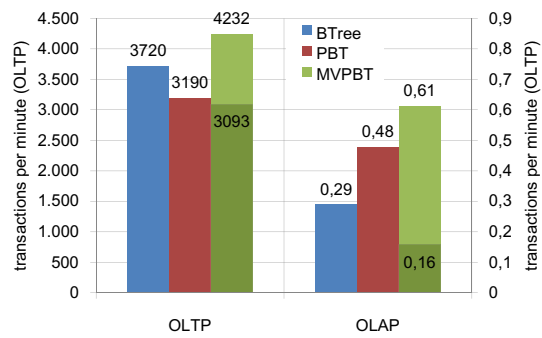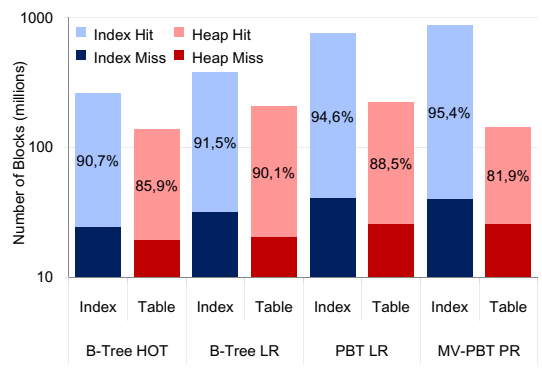
(a) Index Performance under Mixed Workloads (CH-Bnchmark)

(b) Standard vs. Index-Only Visibility-Check for Different Chain Lengths
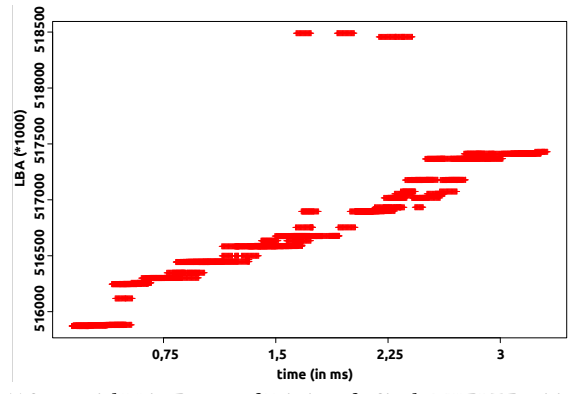
(c) Sequential Write Pattern of Eviction of a Single MV-PBT Partition

(d) Requests / Cache Hit Rate for PostgreSQL Heap-Only Tuples (HOT), Logical(LR) and Physical(PR) references

Figure 12: Index Performance under Mixed Workloads (CH-Benchmark)

eviction time; the average write I/O time is about 1ms. The Y-axis represents the logical block addresses (LBA), i.e. the file system addresses where the blocks of the index file are written. Each red cross indicates the write of a single index node. A horizontal line, therefore indicates a *sequential write*, i.e multiple blocks are written onto neighbouring addresses over time. *Hence the sequential write pattern of MV-PBT.* The horizontal lines in Figure 12c represent *database extents* and result from the database *space allocation strategy*. Each evicted partition comprises leaf nodes allocated in new extents of the index file, allocated at (mostly) adjacent addresses by the file system. The overall *sequential pattern* confirms the sequential append behaviour of MV-PBT.

MV-PBT Buffer Efficiency. Figure 12d shows the fetch requests on index nodes (blue) and base table nodes (red) for an OLTP benchmark. Furthermore, the cache hit-rate is depicted. Requests yielding a cache-hit are displayed in brighter colours than fetches (cache-misses) from secondary storage. The scale of requests is logarithmic. The results are calculated for equal throughput over the test duration and all tables and indices.

PBT and MV-PBT require more requests on index nodes due to partitioning of index records and greater record sizes. Most requests are on buffered nodes, because many queries can be answered in the main memory partition. Index records of new tuple-versions are common to be located there. MV-PBT reduces the requests on base table by up to 40%, because the base table is not required for visibility-check. The version chains are short for this benchmark, for mixed workloads this effect is more weighty.

This can be seen at the reduced cache hit rate on base table nodes in comparison to PBT. Most saved requests on base tables are on new tuple-versions, which are located in main memory.

Partition Filters. Partition-based indices like MV-PBT, PBT or LSM-Trees incur higher lookup and scan overhead than B-Trees, since matching records may exist in older partitions. Hence, the effort of lookups and especially of scans increases with number of index-partitions, since in the worst case every partition has to be traversed. Point lookups can stop partition traversal after finding the first matching record, which is visible to a transaction, since older partitions are guaranteed to contain older records.
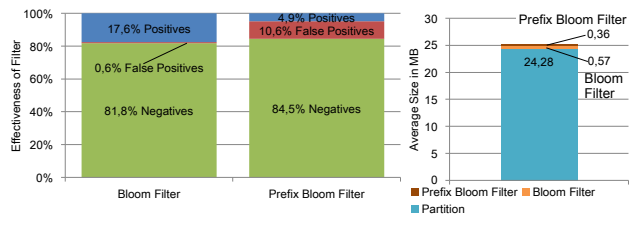

Figure 13: Effectiveness and Size of Partition Filters

Using *Bloom filters (BF)* (Section 4.7) point lookups can skip partitions and increase throughput up to 10% under TPC-C (Figure 14c). Furthermore, *prefix Bloom filters (pBF)* may under certain conditions speedup scans by skipping partitions not matching the range predicate. *pBF* including a fixed set of scan attributes,
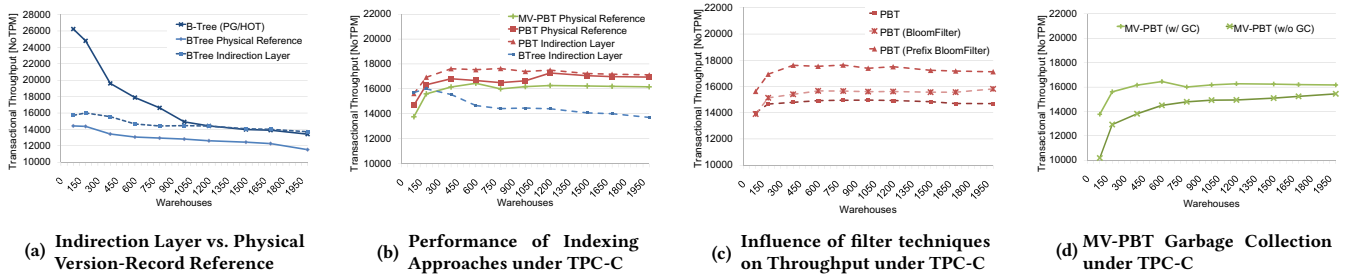
| (a) Indirection Layer vs. Physical Version-Record Reference | (b) Performance of Indexing Approaches under TPC-C | (c) Influence of filter techniques on Throughput under TPC-C | (d) MV-PBT Garbage Collection under TPC-C |

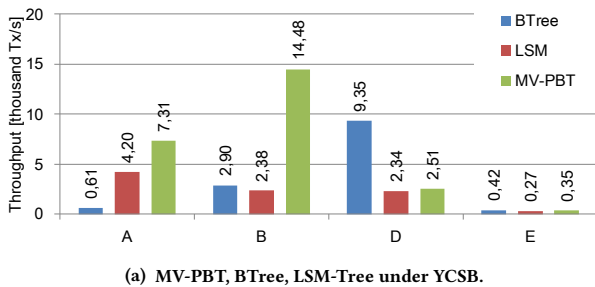**Figure 14: OLTP Performance Evaluation under TPC-C**

increase the throughput by another 10% (Figure 14c). The precision of both Bloom filters is relatively high (Figure 13): the false positives rate is 2% for *BF* and 10% for *pBF*, while the negatives (skipping) rate is approx. 82% for *BF* and 84.5% for *pBF*. The size *BF* and *pBF* is small relative to the partition size (Figure 14c): for a 24MB partition *BF* is 0.57MB, while *pBF* is 0.36MB.

Since index operations only have a fair share of the whole database operations under TPC-C (besides logging, CC and I/O) the above numbers yield moderate performance improvements.
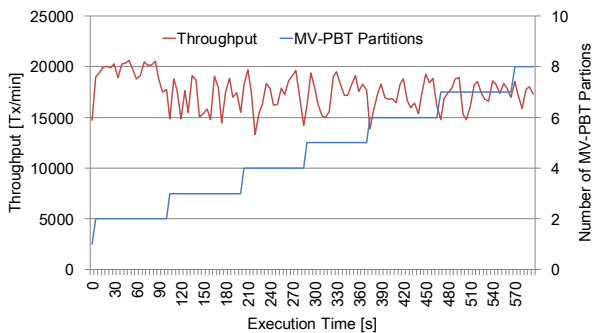
Comparison to LSM-Trees. LSM-Trees [17] are used as workhorse storage structure in many Key/Value stores for large datasets. Today's highly-optimized multi-level LSM-Trees with levelling or tiering resemble MV-PBT as they exhibit an append-behaviour and employ buffered components. We implemented MV-PBT in WiredTiger [3], the high-performance KV-Store of MongoDB. In this experiment we compare MV-PBT to LSM-Tree in WiredTiger under YCSB [7] (Figure 15a). YCSB has been instrumented as follows: a dataset of 100 million keys (approx. 100GB); workloads A (30 mil. requests), B and D(10 mil. req.) and E (2 mil. req.).

(a) MV-PBT, BTree, LSM-Tree under YCSB.

(b) YCSB Throughput (workload A) vs. Number of MV-PBT Partitions

Figure 15: Performance Evaluation under YCSB [7]

Workload *A* comprises 50% read and 50% update requests, which require fast lookups and updates. MV-PBT is approx. 42% faster than LSM-Trees. Each LSM level comprises multiple components, which themselves are small read-optimised BTrees. A search needs to process separate LSM components even though some can be skipped (bloom filters). MV-PBT partition search is faster than LSM component search, since the leaf nodes in each partition are under the same common index. Updates in MV-PBT hit P_N, which accommodates more KV-pairs than the main memory L_0 in LSM-Trees. Workload *B* comprises 95% read and 5% update requests, with *zipfian* distribution. BTree performs random reads, the LSM Tree caches the updates, but has an equal amount of random reads spread over more components. MV-PBT have much lower index maintenance compared to BTrees and place the updates in P_N. The reads are performed with maximum I/O parallelism. Workload *D* comprising 95% read and 5% update requests, which given the *latest* distribution stress the memory components and BTrees performs most of the operations in memory. MV-PBT is marginally better than LSM-Tree. Last but not least, we run workload *E* comprising 95% scans and 5% insert requests. Even though the *scans* are slow under MV-PBT, they outperform LSM-Trees due to the faster search and updates.

Consider, Figure 15b depicting the YCSB throughput (workload A) and the number of MV-PBT partitions over time. The throughput remains stable as the number of partitions increases.

OLTP: comparison of B-Tree alternatives. To establish the baseline we first compare standard PostgreSQL B-Trees (PG/HOT) to B^+-Trees with physical reference and indirection layer on top of append-only storage (SIAS [9, 11]) under TPC-C. In Figure 14a, we show the throughput for different dataset sizes. The buffer cache of the DBMS is fixed to 600MB. B-Tree(PG/HOT) performs well (Figure 14a) as long as the database buffer can accommodate most modifications. Under standard Postgres updates are performed in base tables by Heap-Only Tuples (HOT), i.e. the predecessor version is cached on the same page, on which its successor is located. Therefore the index maintenance effort is low. With growing data sizes (and therefore more modifications), the throughput falls rapidly. Append-based storage and one-point invalidation (SIAS [9, 11]) exhibit a *robust* throughput: (a) *physical references* (Section 3.5) yield lower performance, due to the higher index management overhead; (b) an *indirection layer* reduces index maintenance for insertions and index-key updates, yielding up to 30% better throughput. *With larger datasets (≥ 1200 warehouses) B-Trees with indirection outperform standard PostgreSQL PG/HOT.*

Indexing Approaches under OLTP. In a follow-up experiment, we compare B-Tree with indirection layer (Section 3.5), to

PBT and *MV-PBT* under TPC-C (Figure 14b). PBT and MV-PBT exhibit robust performance, which improves with larger datasets compared to B-Tree. PBT with *indirection layer* exhibits high and robust performance (Figure 14b). PBT with *physical reference* to close the performance gap for larger datasets as the update density decreases decreases with larger datasets. *MV-PBT are slower than PBT under OLTP workloads for several reasons.* First, less MV-PBT index records fit on the same sized P_N, since their sizes are larger because of the version-information (transaction timestamps). Consequently, the number of partitions increases, yielding more I/O. Second, the average version-chain length under TPC-C is short: 1.15/2.18 versions for *customer*/*stock* respectively [9]. Therefore, index-only visibility-checks cannot improve performance significantly. Thus, MV-PBT exhibit 6% lower performance than PBT under TPC-C (Figure 14b). We implemented MV-PBT with an *indirection layer* as well as with *physical references* (Section 3.5). Figure 14b depicts on the performance with *physical references* for brevity, both curves are almost identical. *Therefore, MV-PBT are general enough to be implemented matching the rest of the system design.*

OLTP Garbage Collection. In this experiment (Figure 14d) we quantify the performance effect of MV-PBT partition garbage collection (Section 4.6). It improves performance between 5% and 17% since old invisible versions are purged and need not be processed by scans as well as space is reclaimed letting more index records fit in P_N. The opportunity of improvement under OLTP is however limited by the short average version-chain length: 1.15 versions for *customer* and 2.18 versions for *stock* under TPC-C [9]. With HTAP workloads the amount of 'transient' (short-lived versions visible only throughout the duration of an analytical query) versions increases rapidly as does the effect of garbage collection. Garbage collecting larger amounts transient versions has a major role on the performace improvment of MV-PBT over PBT and B-Tree under mixed workloads (Figure 12a).

6 RELATED WORK

Most popular indexing approaches in database management systems are based on B^+-Trees. Their alphanumeric sorted structure can result in high write amplification for high update rates and visibility-checks require information, that is only located at tuple-versions in base table. PostgreSQL uses Heap-Only Tuples (HOT) 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 held on the same node and can be located by processing the version chain. If a tuple-version become garbage collected, the item is modified to reference the next version. This indirection layer reduces index modifications, but cannot avoid write amplification of index nodes and requires the base table for visibility-checking. Furthermore the write amplification of base table nodes is increased for large datasets. MV-IDX[10] maintains a virtual identifier for each tuple and data nodes for each version as an indirection layer. With Snapshot Isolation Append Storage (SIAS)[11] write amplification on base tables is reduced in comparison to HOT, but index management operations can cause a high write amplification and base table nodes are still required for visibility-checking[21]. LSM-Trees[18] reduce write amplification due to collecting modifications in main memory components, but there is no concept for managing tuple-versions and perform an index-only visibility-check[21]. Time-Split B-Trees [16] and Multiversion B-Trees [5]

are able to separate index records of old tuple-versions from current dataset and to perform an index-only visibility-check, but maintenance operations are complex and can cause a high write amplification of index nodes[21].

7 CONCLUSION

In the present paper we introduce MV-PBT as an approach to multi-version indexing. An MV-PBT is an extension of a B-Tree, where an artificial leading column is prepended to the search key of each index record and index records are placed in a buffered index partition, which if full gets evicted and appended to persistent storage. MV-PBT is version-aware, since index records contain version-information and allow for index-only visibility check. This is particularly beneficial for HTAP workloads since long chains of transient versions exist due to the mix of short-lived updating transactions and long-running queries. Furthermore, MV-PBT exhibit a sequential write pattern due to the concept of partition, which leads to less write-amplification and better utilization of modern storage technologies. Under mixed workloads (CH-Benchmark) MV-PBT doubles the analytical throughput 2x, while improving the transactional throughput by 15%.

REFERENCES

[1] 2019. *Database Test Suite*. https://sourceforge.net/projects/osdldbt/files/dbt2/
[2] 2019. *Oltpbench*. https://github.com/oltpbenchmark/oltpbench/
[3] 2019. *WiredTiger (MongoDB)*. http://www.wiredtiger.com
[4] R. Bayer and E. McCreight. 1970. Organization and Maintenance of Large Ordered Indices. In *Proc. SIGFIDET (SIGMOD) 1970*. 107–141.
[5] Bruno Becker, Stephan Gschwind, and et al. 1996. An Asymptotically Optimal Multiversion B-tree. *The VLDB Journal* 5, 4 (Dec. 1996), 264–275.
[6] Richard Cole, Florian Funke, Alfons Kemper, and et al. 2011. The Mixed Workload CH-benCHmark. In *Proc. DBTest '11*. Article 8, 6 pages.
[7] Brian F. Cooper, Adam Silberstein, and et al. 2010. Benchmarking Cloud Serving Systems with YCSB. In *In Proc. SoCC2010*.
[8] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (Dec. 2013), 277–288.
[9] Robert Gottstein. 2016. *Impact of new storage technologies on an OLTP DBMS, its architecture and algorithms*. Ph.D. Dissertation. TU, Darmstadt.
[10] Robert Gottstein, Sergej Hardock, Ilia Petrov, and Alejandro Buchmann. 2014. MV-IDX: Indexing in Multi-version Databases. In *Proc. IDEAS 2014*. 142–148.
[11] Robert Gottstein, Ilia Petrov, and et al. 2017. SIAS-Chains: Snapshot Isolation Append Storage Chains. In *ADMS@VLDB*.
[12] Goetz Graefe. 2003. Partitioned B-trees - a user's guide. In *Proc. BTW*. 668–671.
[13] Goetz Graefe. 2003. Sorting And Indexing With Partitioned B-Trees. In *CIDR*.
[14] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, and et al. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *Proc. SIGMOD 2016*. 1307–1318.
[15] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proc. ICDE 2013*. 302–313.
[16] David Lomet and Betty Salzberg. 1990. The Performance of a Multiversion Access Method. In *Proc. SIGMOD 1990*. 353–363.
[17] Chen Luo and Michael J. Carey. 2019. LSM-based storage techniques: a survey. *The VLDB Journal* (19 Jul 2019).
[18] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385.
[19] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid Transactional/Analytical Processing: A Survey. In *Proc. SIGMOD 2017*. 1771–1775.
[20] I. Petrov, R. Gottstein, and S. Hardock. 2015. DBMS on modern storage hardware. In *Proc. ICDE 2015*. 1545–1548.
[21] Christian Riegger, Tobias Vincon, and Ilia Petrov. 2017. Multi-version Indexing and Modern Hardware Technologies A Survey of Present Indexing Approaches. In *Proc. iiWAS 2017*. 266–275.
[22] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, and et al. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proc. SIGMOD 2018*. 473–488.
[23] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-memory Multi-version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (March 2017), 781–792.