Information Systems 125 (2024) 102403

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Storage Management with Multi-Version Partitioned BTrees

Christian Riegger [*], Ilia Petrov

Data Management Lab, Reutlingen University, Alteburgsr. 150, Reutlingen, 72762, BW, Germany

ARTICLE INFO

Keywords:
Storage engine
Storage management
Index management
Append storage
Time-travel query processing

ABSTRACT

Modern persistent Key/Value-Stores operate on updatable datasets — massively exceeding the size of available main memory. Tree-based key/value storage management structures became particularly popular in storage engines. B$^+$-Trees allow constant search performance, however write-heavy workloads yield inefficient write patterns to secondary storage devices and poor performance characteristics. LSM-Trees overcome this issue by horizontal partitioning fractions of data — small enough to fully reside in main memory, but require frequent maintenance to sustain search performance.

To this end, firstly, we propose Multi-Version Partitioned BTrees (MV-PBT) as sole storage and index management structure in key-sorted storage engines like Key/Value-Stores. Secondly, we compare MV-PBT against LSM-Trees. The logical horizontal partitioning in MV-PBT allows leveraging recent advances in modern B$^+$-Tree techniques in a small transparent and memory resident portion of the structure. Structural properties sustain steady read performance, even on historical data, and yield efficient write patterns as well as reduced write-amplification.

We integrate MV-PBT in the WiredTiger key/value storage engine. MV-PBT offers an up to 2x increased steady throughput in comparison to LSM-Trees and several orders of magnitude in comparison to B$^+$-Trees in a YCSB workload. Moreover, MV-PBT exhibits robust time-travel query performance and outperforms LSM-Trees by 20% and B$^+$-Trees by an order of magnitude.

1. Introduction

High performance persistent key-sorted No-SQL storage engines became the workhorse systems for online data-intensive applications. Such engines exist as standalone K/V-Stores (Key/Value Stores) [1,2] as well as in DBMS integrated as storage engines [3–5]. Obviously, backing tree-based K/V storage management structures – *i.e.* B$^+$-Trees [6], LSM [7,8] and derivatives [3,9] – natively enable necessary advanced lookup operations beside equality search, *e.g.* key prefix or inclusive and exclusive range searches, with (nearly) constant logarithmically scaling performance characteristics. Continuous modifications require special care to preserve constant performance characteristics and mentioned search features. Although B$^+$-Trees offer constant search performance to data in main memory and on secondary storage devices, modifications yield inelastic performance characteristics. LSM-Trees sacrifice properties of a single tree structure to overcome this issue by buffering modifications in a fraction of main memory, typically tree-based components, and leveraging flash-based secondary storage device characteristics on eviction and necessary background merge operations.

Flash technology in SSD secondary storage devices offers specific individual characteristics. I/O operations are executed against multiple independent storage units of the Flash-SSD such as (chips, dies, planes) that are connected over independent channels SSD controllers, thus resulting in a high internal parallelism and I/O-performance [10–12]. However, reads perform an order of magnitude better than writes, yielding an asymmetric read/write I/O behavior. While read performance is nearly identical for random and sequential access patterns, write I/O is preferably sequentially performed [13]. Furthermore, pages are only overwritten out-of-place (so called erase-before-overwrite), mandating possibly preemptive background garbage collection [14,15] to mitigate the impact of very slow flash erases.

Focus on Storage Amplification Factors. Generally, tree-based storage management structures manage *records*, *e.g.* a set of key–value pairs with a size of several hundred *bytes (B)* respectively, in a structure of *nodes*, which possibly have a fixed size that is a multiple of a secondary storage device's block-size, ranging from few *kilo bytes (kB)* up to *mega bytes (MB)*. Consequently, there is a discrepancy in size and costs of logically required and physically transferred data for different database operations and storage structures. For instance, if an $100B$ key–value pair is to be read, its logical size is $100B$. However, searching its respective *record* in the structure of *nodes* typically yields a sequence

* Corresponding author.
E-mail address: christian.riegger@reutlingen-university.de (C. Riegger).

https://doi.org/10.1016/j.is.2024.102403
Received 14 February 2023; Received in revised form 20 March 2024; Accepted 12 May 2024
Available online 15 May 2024

of physical block reads from secondary storage devices. Hence, several *kB* or *MB* are physically read in order to logically get the 100*B* key–value pair. On the other hand, a logical insertion of an 100*B* key–value pair yields a subsequent physical write of a set of *nodes*. Thus, physical read/write I/O respectively causes *read-amplification (RA)* or *write-amplification (WA)* for operations (*e.g.* `get` or `put`) on logical key–value pairs. Moreover, *space-amplification (SA)* describes the ratio of physically and logically required space per modifying information. Smaller ratios indicate less resource consumption and facilitate better storage characteristics.

Multi-Version Storage enables Time-Travel Capabilities. The ability to query the dataset for a particular point in history is very useful in enterprise applications [16]. Several storage engines already maintain multiple versions of a logical tuple for certain points in time in order to perform multi-version concurrency control (MVCC) and snapshot isolation (SI). Nevertheless, obsolete version records get removed by background garbage collection. Major difference in time-travel query processing is that version records principally never become obsolete. Especially in mixed workloads, robust transaction processing and query performance is challenging with increasing version chain lengths.

Contributions. The main contribution of this paper targets the use of *Multi-Version Partitioned BTrees* (MV-PBT) as the sole storage structure key–value storage engines. Detailed contributions are as follows [17]:

- Reducing write-amplification in append-based storage management with MV-PBT by sequential write of saturated partition managed nodes.
- Transparent internal partition management and atomic partition switch operations without schema maintenance requirements.
- Single root node as entry point in the B^+-Tree structure allows to leverage logarithmic capacity and buffering/caching behavior for commonly traversed inner nodes.
- Reduction of merge-triggered write-amplification and accompanying pressure on secondary storage devices by *Cached Partitions*.
- Leveraging scalable in-memory optimizations and compression techniques of B^+-Tree structures for massive amounts of data in a very hot fraction.
- Prototypical implementation and experimental evaluation in WiredTiger [2], which provides competitive B^+-Tree and LSM-Tree implementations.

The present paper introduces the following novel aspects:

- Implications on secondary indexing MV-PBT storage engines.
- Robust latencies in time-travel query processing with experimental evaluation in WiredTiger.

Outline. First, we give an overview of common tree-based storage structures as well as recently introduced aspects of *Multi-Version Partitioned BTrees (MV-PBT)* [17,18]. We present an architectural overview of MV-PBT in Section 3. Sections 4 and 5 focus on reduction of write-amplification by data skipping and fast retrieval in a horizontally partitioned structure and considering defragmentation only as a result of garbage collection. gives an exemplary illustration of implications on secondary indexing MV-PBT. Section 7 focus on time-travel query processing capabilities of MV-PBT. We evaluated the storage management structures in the homogeneous storage engine WiredTiger 10.0.1 in Section 8 and conclude in Section 9.

2. Related Work

Most popular key-sorted storage and index management structures, including LSM-Trees [7,8], derivatives [3,9] as well as the proposed approach *Multi-Version Partitioned BTrees (MV-PBT)* [17,18], are based on B^+-Trees [6]. Hence, we now provide a brief overview:

B^+-Trees and derivatives achieve a constant logarithmically scalable search performance, since root-to-leaf traversal operations depend on their height — even in case of massive amounts of stored data records. Commonly used inner nodes of traversal paths allow fast access to data in leaf nodes with few successive read I/O. However, B^+-Trees are potentially vulnerable in case of heavy modifications. Whilst insertions, updates and deletions of records possibly facilitate steady throughput in main memory by optimized and highly scalable maintenance procedures [2,3], massive amounts of maintainable key-sorted data yield random write I/O and high write-amplification on secondary storage devices once modifications get persisted on eviction of volatile and modified (‘*dirty*’) buffers. In order to preserve strict lexicographical sort order of records, maintenance operations cause cascading node splits, whereby blank space is created to accommodate additional separator keys in inner nodes and records in leaves in the designated arrangement. As a result, sub-optimally filled nodes reduce cache efficiency and contained information is written multiple times, yielding a high write and space-amplification. Furthermore, read I/O on secondary storage devices of partially filled nodes lead to high read-amplification. Therefore, for massive amounts of data, B^+-Trees become write-intensive, even in case of proportionately few modifications.

Thus the following **problems** emerge:

- low benefit from main memory optimizations, since nodes are frequently evicted.
- low cache efficiency and high read-amplification due to partially filled nodes.
- massive space and write-amplification on secondary storage devices.

Alternatively, LSM-Trees are optimized for high insert/update rates and yield a sequential write pattern, since modifications are buffered in tree-based LSM components in main memory. Components get frequently switched, merged and evicted to persistent secondary storage devices. Generally, background merge operations (so called compactions) counteract the data fragmentation and increased read and search effort, however this behavior also increases its write-amplification. Several approaches in merge policies [19–21] and reduction of read-amplification [22–25] have been introduced. Certainly, flash allows high internal parallelism and multiple reads of parallel traversal operations. Nevertheless, since components are *separate structures*, they effectively leverage neither caching effects on traversal nor logarithmic capacity capabilities per height of B^+-Trees. Moreover, creation of new components on switch procedure is not transparent to the storage engine and relies on high-level maintenance of the database schema. Finally, due to append-based record replacement technique in LSM, key uniqueness is assumed, wherefore the application in storage engines of DBMS with non-unique indexes is complicated.

The **challenges in LSM** are defined as follows:

- inefficient caching behavior of decoupled components require frequent merges and yield considerable write-amplification.
- hence, high internal parallelism of flash is not leveraged for read operations.
- components are non-transparent for further layers of a storage engine.
- non-unique indexing requires additional care.

Multi-Version Partitioned BTree (MV-PBT) serves as sole storage and index management structure in KV-storage engines. MV-PBT is based on Partitioned BTrees (PBT) [26], an enhancement of a traditional B^+-Tree. Recent publications introduced (MV-) PBT as a highly scalable indexing structure in DBMS with multi-version concurrency control (MVCC) and massive index update pressure [18,27,28], combining and extending the beneficial properties of B^+-Trees and LSM-Trees. (MV-) PBT relies on manipulation of an artificial leading key column of

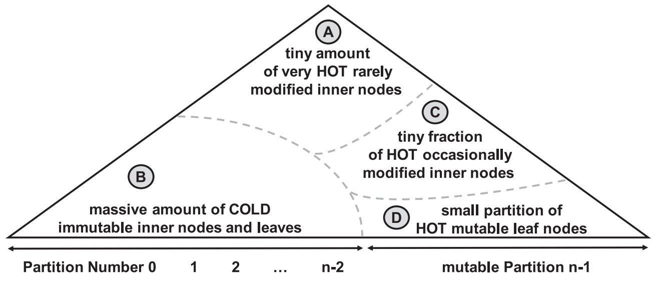


Fig. 1. Schematic representation of MV-PBT's logical organization. Logical horizontal partitioning and buffer replacement policy yield a hot/cold separation within one single tree structure and ultimately enables a sequential write pattern of whole partitions.

every record — the partition number; and exploiting the regular lexicographical structure of B$^+$-Trees for partition management due to several purposes. Similar to component management in LSM-Trees, partitions are adopted for write-optimization (compare [17,27,28]), however, within one single tree-structure with commonly used nodes. Therefore, modifications are collected in a dedicated partition, *e.g.* with the *most recent partition number*, located in the *MV-PBT-Buffer* in main memory, and evicted in a sequential write. Moreover, partition management additionally enables maintenance of several version records and inherent index-only visibility checks [18], *e.g.* in the context of *MVCC* and *SI*, and serve as storage management structure [17]. This paper extends the focus on MV-PBT as sole storage management structure [17] by the contexts of *additional access paths*, *i.e.* MV-PBT as sole storage and index management structure, and *time-travel capabilities* in KV-storage engines.

3. Architecture of Multi-Version Partitioned BTrees

Multi-Version Partitioned BTree (MV-PBT) as an append-based and version-aware storage and indexing structure relies on well-studied algorithms and structures of traditional B$^+$-Trees, with which they share many characteristics and areas of application. Therefore, MV-PBT is able to adopt and even leverage modern B$^+$-Tree techniques.

Structural Overview. The proposed approach facilitates straightforward horizontal partition management within one single B$^+$-Tree structure in order to keep a very hot mutable fraction of leaves in fast main memory (Fig. 1 (D)), *i.e.* the *MV-PBT-Buffer* includes the *most recent partition* leaves and is kept apart from the regular buffer replacement policy (compare Fig. 1 nodes in (A,B,C)). Reaching a certain dirty memory footprint (*MV-PBT-Buffer threshold*) initiates an atomic partition switch operation, which asynchronously finalizes in a sequential write of dense-packed cleaned data in leaves and referring inner nodes, in order to interference-freely absorb ongoing modifications. The entire process is explained in a separate paragraph later in this section. Since partitions are principally defined by the existence of associated records, they appear and vanish as simply as inserting or deleting records [26], however, auxiliary meta data structures allow a massive speed-up of operations. Append-based structures allow modifications of already persisted data by out-of-place replacement. MV-PBT enhances this behavior by additional record types, which allow internal indexing and non-uniqueness of data and enables native B$^+$-Tree-like indexing features. Moreover, maintenance of multiple version records of the same data items mandate the adoption of multi-version capabilities in terms of transaction timestamps and multi-version (MVCC) visibility checking. *Low write-amplification, sequential writes of dense-packed nodes, commonly utilized inner nodes with one single root as entry point, parallelized multi-partition search operations as well as multi-version indexing capabilities make MV-PBT superior as sole storage and index management structure in storage engines.*

MV-PBTs Auxiliary Data Structures information is entirely contained in the B$^+$-Tree structure. For instance, the mutable most recent partition number (compare Fig. 1 (D) *mutable Partition* and Fig. 2 *max_pnr*) could be identified by searching the rightmost record in the tree structure. Since cached information is frequently required and its memory footprint is very low, auxiliary data structures are cached in RAM (an excerpt is depicted in Fig. 2). MV-PBT data structures require neither locking for any atomic operation, nor additional logging of modifications, since the lightweight information is completely recoverable from basic B$^+$-Tree by a scan operation. All information of horizontal partitioning is anchored within the tree structure, *i.e.* horizontal partitioning is transparent to further storage engine modules.

Multiple MV-PBT may exist within a storage engine, which commonly share the *MV-PBT-Buffer threshold* for leaves of respective *mutable most recent partitions* (Fig. 2 *Global Meta Data* — `buffer_share`). The MV-PBT Meta Data belongs to a specific relation in the schema, indicated by `relation_id` in Fig. 2. Its most recent partition number (`max_pnr`) is frequently required to determine record prefixes as well as for atomic switching operation (`is_switching`). An MV-PBT comprises of several valid partitions, which contain a set of meta data like its respective *partition number* (`pnr`), an indicator for its persistent storage (`synced`), the number of comprised records (`n_records`), the size of associated leaves in the MV-PBT-Buffer (`dirty_leaf`) or specific partition type characteristics (`type`). Finally, *auxiliary filter structures* for point and/or range queries are referenced; *e.g.* fence keys, (prefix) bloom filters or hybrid point and range filters [22–24].

Partition Number Prefixes are prepended to each record key with the central scope of leveraging lexicographical sort capabilities of B$^+$-Trees in order to achieve a logical horizontal partitioning (compare Fig. 3.a arrangement of record *Maya* in B$^+$-Tree and MV-PBT). Partition numbers could be of any comparable data type, *e.g.* 2 or 4-byte integers, and might are maintained in an artificial leading key column [26].

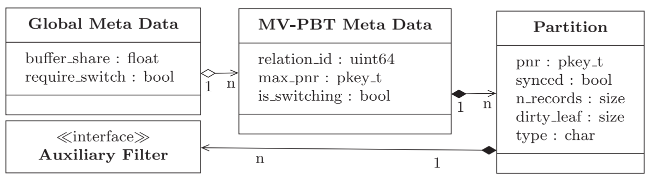


Fig. 2. Auxiliary recoverable MV-PBT data structures.

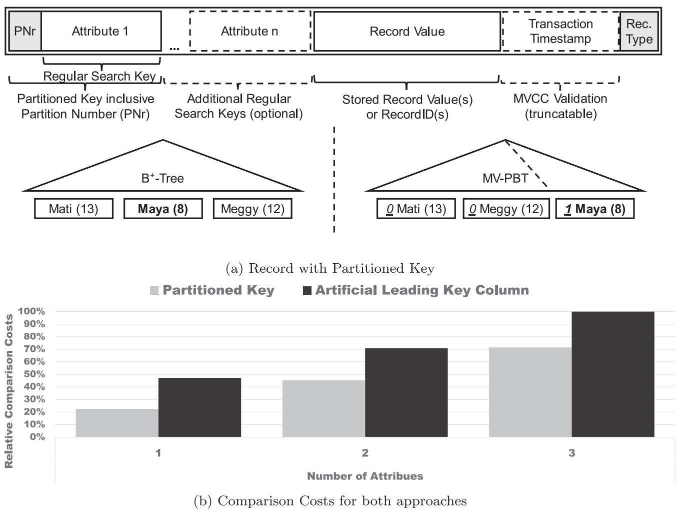
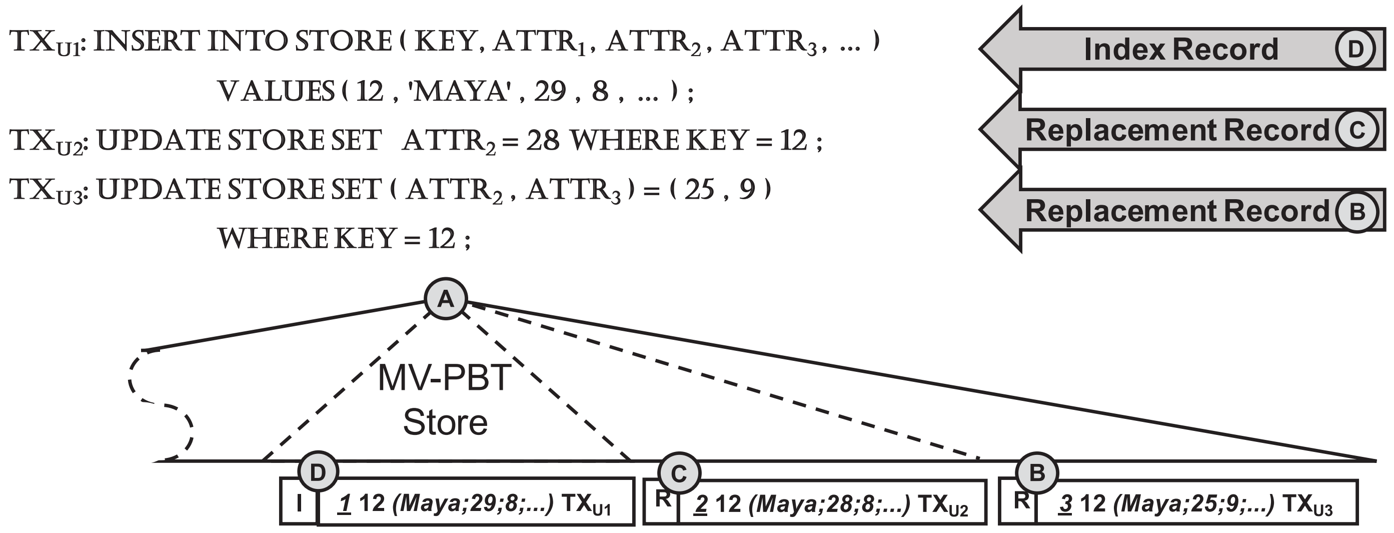


Fig. 3. Horizontal partition maintenance with Partitioned Keys.



TXU1: INSERT INTO STORE (KEY, ATTR1, ATTR2, ATTR3, ...)
VALUES (12 , 'MAYA' , 29 , 8 , ...) ;
TXU2: UPDATE STORE SET ATTR2 = 28 WHERE KEY = 12 ;
TXU3: UPDATE STORE SET (ATTR2 , ATTR3) = (25 , 9)
WHERE KEY = 12 ;
Index Record D
Replacement Record C
Replacement Record B
A
MV-PBT Store
D
I 1 12 (Maya;29;8;...) TXU1
C
R 2 12 (Maya;28;8;...) TXU2
B
R 3 12 (Maya;25;9;...) TXU3

Fig. 4. Example of append-based out-of-place record maintenance in MV-PBT.

However, combining the partition number and the first record key attribute in a *partitioned key* type (compare *Attribute 1* in Fig. 3.a) enables cache efficient comparison of co-aligned attributes as evaluated in Fig. 3.b — *artificial leading key columns cause increased costs of one additional comparison*. Additional storage costs are negligible due to prefix truncation techniques. *Partitioned keys* are simply allocated when setting search keys and their prefix becomes hidden by returning an offset in the leading key attribute in order to retain transparent horizontal partitioning.

Multi-Version Capabilities match the out-of-place replacement in MV-PBT. Multi-Version Concurrency Control (MVCC) with Snapshot Isolation (SI) is a common technique to enable high transactional parallelism in storage engines, since readers and writers are not mutually blocking as each transaction operates on a separate snapshot of data. Therefore, multiple version records of one logical tuple are maintained in a version chain — each is valid for a different period in time.

According to the example in Fig. 4, MV-PBT adopts a beneficial new-to-old ordering approach [29] of physically materialized version records with out-of-place update scheme and one-point invalidation model [18,30] — *i.e.* predecessor versions remain unchanged on modification, whereas write-amplification is massively reduced. For illustration purposes, in Fig. 4, a predecessor record (D) requires no modifications on updates due to replacement by out-of-place successor version records (C) and (B). Successor version records are augmented with the current transaction timestamp (*e.g.* TX_{U2} and TX_{U3}, which may become truncated on eviction to secondary storage devices, whenever no preceding snapshot is active) and are inserted in the most recent partition in the MV-PBT-Buffer. Thereby, it is possible to maintain multiple version records in several succeeding partitions (compare example in Fig. 4 or a single modifiable partition, *e.g.* as separate record [18] or in-memory update lists [2,17]. Given the logical search succession in MV-PBT from new-to-old, *e.g.* partition traversals from (A) to (B), (A) to (C) and (A) to (D) in Fig. 4, transaction snapshots identify their visible version record and skip others, based on the augmented transaction timestamps. Since record data values are physically materialized in each version record, identified records are directly applicable.

Record Types in MV-PBT feature all operations over logical tuple lifecycle without modifying predecessor version records. During lifetime, it gets created, modified and deleted while it is frequently read.

- *Regular Records* declare the begin of the life-cycle, hence there is no predecessor version. Its transaction timestamp is applied by the inserting transaction and indicates its validation.
- *Replacement and Anti-Records.* Replacement records indicate a new record value on update. Its timestamp invalidates its predecessor as well as validates itself. Replacement Records are also applied on modifications to the record key, however, invalidation requires an *Anti Record* with the predecessor key attribute values and the current transaction timestamp for invalidation. Replacement Records as well as Anti Records mostly store the predecessor's value for logical tuple assignment as needed in non-uniqueness index management constraints. However, modifications to the key attributes and non-uniqueness indexing constraints with index-only visibility checks [18]

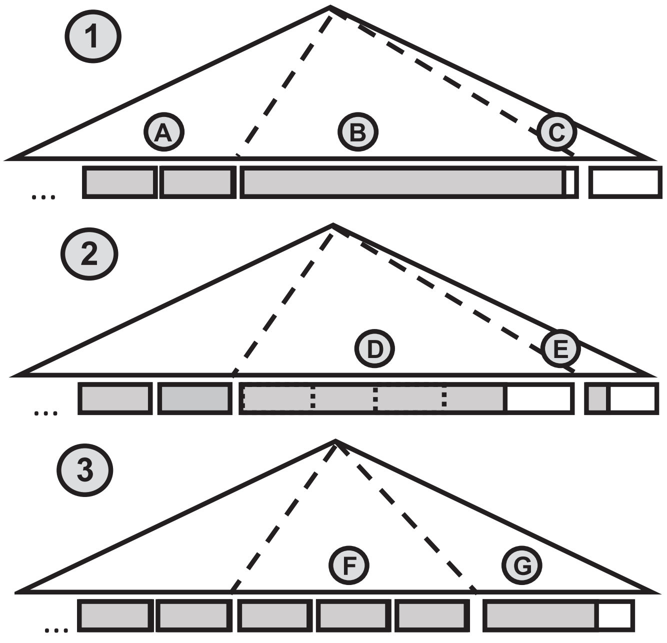


Fig. 5. Conversion to a *(iv) defragmented and dense-packed disk layout* and inclusive *(vi) performance of a sequential write of leaves and referring inner nodes* with modern B^{+}-Tree techniques. (1) After atomic partition switch (stage *iii*), an MV-PBT consists of (A) persistent, (B) a victim and (C) a most recent partition. Internal nodes and leaves of the victim partition delay maintenance effort (*e.g.* split operations) by flexible page size until a reconciliation process (2.D). The (E) most recent partition consumes ongoing modifications. Finally, (3) the (F) victim partition is sequentially written to secondary storage and (G) is the only memory mapped partition.

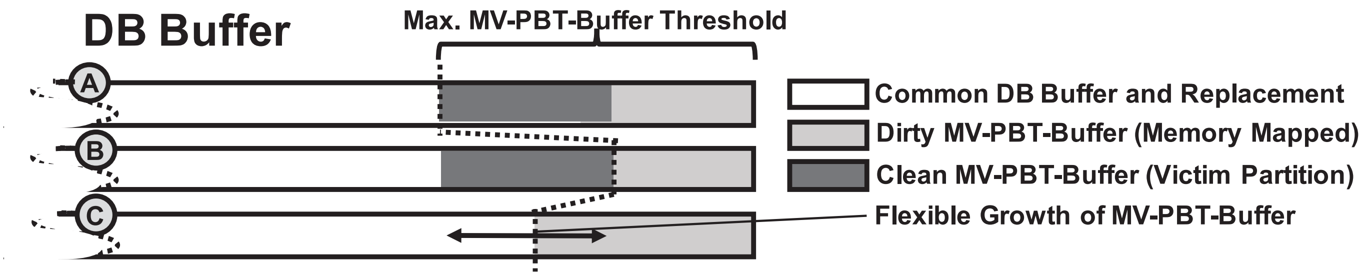

DB Buffer
Max. MV-PBT-Buffer Threshold
A
B
C
Common DB Buffer and Replacement
Dirty MV-PBT-Buffer (Memory Mapped)
Clean MV-PBT-Buffer (Victim Partition)
Flexible Growth of MV-PBT-Buffer

Fig. 6. Flexible MV-PBT-Buffer Threshold allows cache preserving handover of a clean Victim Partition from (A) the MV-PBT-Buffer to (B) a common buffer replacement policy and (C) flexible growth up to a max. MV-PBT-Buffer Threshold.

allow MV-PBT to serve as sole storage [17] and index management structure in storage engines.

- Finally, *Tombstone Records* are inserted on deletion of a logical tuple. Major difference to Anti Records is, that successor version records are impossible.

Atomic Partition Switch and sequential write of dense-packed leaves and referring inner nodes bring a leading edge in MV-PBT. The whole procedure consists of several partially parallelizable stages (*i to vii*) [17], which are outlined in the following. After *(i) determination of switch requirement* by a certain dirty buffer threshold in the MV-PBT-Buffer, a *(ii) valuable MV-PBT victim partition* is selected for eviction. Contrary to LSM-Trees, MV-PBT partitions become immutable and switched by *(iii) atomically incrementing the most recent partition number* (`max_pnr`) in the meta data, since the required B$^+$-Tree structure is already existent and logged anyways.

However, records are not yet in their final *(iv) defragmented and dense-packed disk layout*, since structure modifications are the result of a randomly inserting workload. One approach to avoid expensive partition-internal structure modifications (*i.e.* node merges) for dense-packing is to simply re-inserting the still valid contents in their final arrangement by manipulating the partition number in a bulk load operation [18]. B$^+$-Trees allow efficient split policies to support high fill factors by this operation. Finally, visibility characteristics of both partitions are swapped and the randomly grown source partition gets cropped from the tree. Another approach is to leverage modern B$^+$-Tree techniques. In order to avoid structure modifications, referenced main memory nodes are allowed to flexibly grow and finally get divided and structured in the disk layout in a reconciliation process (depicted in Fig. 5).

Auxiliary *(v) filter structures* are generated as a natural by-product of defragmentation and dense-packing, since records are accessed anyways. Whenever (a fraction of) leaf nodes obtained their final layout, it is possible to *(vi) perform a sequential write of leaves and referring inner nodes* by traversing the tree structure and following the sibling pointers — yielding a bottom-up sequential write of volatile (*dirty*) nodes, level by level. Finally, the persisted (*clean*) leaves are *(vii) passed to the regular replacement policy* in order to sustain a constant buffer factor and memory footprint (Fig. 6).

Basic Operations in MV-PBT are based on a regular B$^+$-Tree — *i.e.* they have logarithmic complexity. Every modifying operation is treated as an insertion of a record of a respective type [17,18,27,28]. Thereby, the current transaction timestamp is set for validation in visibility checks — and one-point-invalidation of conceivable predecessors, respectively, which can be located in a preceding or the current partition. However, due to the partitioned key, each modifying operation is performed in the most recent partition in main memory. This is also valid in case of concurrent partition switch by overwriting the partition number of an insertion record key and immediate re-traversal from root. Additional constraint support is very uncommon in pure storage management since records are typically overwritten by blind insertions, however, this is facilitated by MV-PBT in preceding equality search operations.

Equality and range search operations perform root-to-leaf traversals of a (sub-)set of partitions by manipulation of the partition number in the partitioned search key. Partitions are preselected by auxiliary filter structures. Logically, partitions are searched in reverse order from the most recent to the lowest numbered one. Based on the selectivity of a query, partitions may be sequentially processed or by parallel traversals in a merge sort operation [17]. In case of equality searches, sequential processing allow minimal read-amplification, contrary, sorted range searches favorably adopt the merge sort approach, whereby multiple cursors are applied and get individually moved and returned to a higher level merge sort cursor. Thereby, record transaction timestamps are checked for visibility to a transaction snapshot. Based on a regular visibility check, invisible and invalidated records are skipped, invalidating records are remembered for exclusion of occurring predecessors (which are subsequently accessed) and matching records are returned [18].

4. Cached Partition: Stop Re-Writing valid Data

MV-PBT introduces a logical horizontal partitioning within one single tree structure in order to leverage characteristics of secondary storage devices [17,27–29]. This data fragmentation influences the search operations in different ways. Obviously, several possible storage locations of a requested record implies additional search effort. Actually regular B$^+$-Trees incur increased search costs in randomly grown structures, due to diminishing cache efficiency of partially filled inner nodes. Contrary, LSM-Trees keep a read-optimized layout within each component, however, multiple entry points and referenced inner nodes are neither commonly cached nor leverage logarithmic capacity [18, 29]. LSM-Trees counteracting increased search effort with background merge operations, whereby write-amplification of still valid data is increased [17].

MV-PBT preserves a read-optimized and cache-efficient layout for immutable nodes (Fig. 1.B) with one commonly shared entry point and referenced inner nodes (Fig. 1.A) which are subjecting to an optimal fill factor, since append-based behavior of referenced data allows efficient split policies (equal to bulk loads). As outlined in Section 3 (*Atomic Partition Switch and sequential write*), mutable inner nodes and leaves (Fig. 1.C and 1.D) are a hot fraction which sustains maintenance operations of the random workload, however, modern B$^+$-Tree techniques allow main memory efficient delay of maintenance operations. Since the small fraction of inner nodes is commonly used, they are well cached, so that a large portion of the parallel traversal operations is performed without read latencies from secondary storage devices. Successive read I/O in multiple partitions leverage parallelism in flash persistent storage. Moreover, search performance in MV-PBT relies on data skipping by auxiliary filter structures. As a combined result, MV-PBT is able to sustain comparable search performance for higher fragmentation as in LSM-Trees [17].

However, variety of auxiliary filter structures imply caching and probe costs as well as massive amount of traversal operations result in high read I/O costs and shrink performance due to growing fragmentation. Instead of adversely re-writing still valid data records in a consolidated arrangement, due to asymmetry of flash and write-amplification, MV-PBT introduces *Cached Partitions* [17]. They are an internal index partition, whose records reference a preceding partition, containing the latest version record of a logical tuple in a lexicographical sort order. Several Cached Partitions may exist for a different subset of small partitions and are cyclically created while the MV-PBT evolves. Cached Partitions are the result of a background merge sort of contents in several immutable lower numbered partitions with the respective partition number as value or the contents of several preceding Cached Partitions. Background merge sort results are bulk inserted in an '*invisible*' partition while proceeding, can be paused and continue without wasting work and become finally visible by an atomic status switch.

Since a subset of partitions is fully indexed in a Cached Partition, a subsequent search operation is able to traverse the subset on the commonly cached path as needed, based on the results of the internal partition index. Cached Partitions assume responsibilities of auxiliary filter structures and allow to exclude the subset of indexed partitions from the regular logical search succession, whereby comparison costs in an internal merge sort are reduced — the effort is focused on non-indexed and Cached Partitions. Furthermore, cached index records are very space and cache efficient in the search process, since they consist of the key and one partition number (*e.g.* 2 or 4-byte integer) in a dense-packed arrangement.

5. Garbage Collection and Space Reclamation

Datasets and tuple values evolve over time. Storage management structures with out-of-place update approaches allow beneficial sequential write patterns and low write-amplification, however, invalidated predecessor record versions remain existent on update. Search operations are able to exclude invalid version records from the result set, though visibility checking entail additional processing. Furthermore, version records which are not visible to any active transaction snapshot entail space-amplification and additional storage costs.

In MV-PBT, additional search costs due to fragmentation by horizontal partitioning is well covered by Cached Partitions for insertion of new tuple version records. However, modifications to logical tuple values leave persisted obsolete version records behind, yielding space-amplification [17]. Ideally, obsolete version records are discarded as part of the dense-packing phase on partition switch, however, many version records become invalidated after they were persisted. For the only reason of space reclamation, MV-PBT occasionally performs a garbage collection (GC) process. Similar to the creation of a Cached Partition, GC is performed by a background merge sort and bulk load operation in a not yet visible partition. Certainly, the stored record value is the regular value of the most recent record version of a tuple. As well, the GC process can throttle and continue without wasting work, since the partition is not yet accessible for querying. After the successful completion, the partition becomes visible and the records of purified preceding partitions become invalidated. Once every active search operation finished, the purified partitions are cropped from the tree structure by an efficient range truncation [2].

6. Implications on Secondary Indexing MV-PBT

Storage management with a key-searchable structure like MV-PBT facilitates additional purposes in K/V-Stores. Primary key indexing on tuple records is feasible without the effort of auxiliary access path maintenance. More to the point, K/V-storage engines (like MyRocks [31] or MongoRocks [32] to name but a few) necessitate support for secondary indices [31,32]. However, efficient searches and scans on various tuple attributes require additional secondary index structures — preferably equipped with index-only visibility checks for optimal selectivity in multi-version scenarios and minimal write-amplification maintenance costs [18].

Logical Referencing of Version Records. MV-PBT as storage management structure maintains modifications to the dataset by insertions of fully materialized version records in the most recent partition in main memory [17]. Storage locations of these version records are temporarily volatile until a *defragmented and dense-packed disk layout* is established in a *victim partition*, due to native B$^+$-Tree ordering capabilities in modifiable partitions. Finally, each version record is addressable by *physical referencing* with page and slot numbers. Nevertheless, since secondary indexes directly reference all records in the entire dataset, physical referencing is neither feasible without additional effort nor necessary especially whenever modifications are performed in the main store.

MV-PBT exhibits constant search performance for each version record by a single traversal operation, since its *partition number* stays unchanged. *Logical referencing* is a sustainable solution for secondary indexing. Index records are formed by *search key attribute values* and a *record value*, which references a version record in the MV-PBT storage management structure by its entire search key. Version-oblivious secondary indexes do not gain benefit from specific version record keys and preferably reference logical tuples (without partition number), wherefore a regular search operation is performed in the MV-PBT storage management structure. *On the contrary, version-aware secondary indexes reference version records (including the partition number)*.

Index-Only Visibility Checks. MV-PBT storage management structures are accessed by various additional access paths with logical reference. Nevertheless, arbitrary index approaches are not always able to identify version records of logical tuples, which are visible to a transaction snapshot. Hence, a regular search succession with visibility check in the main store is required from new-to-old, which might cause traversal operations in several partitions for each index record that matches the search predicates. These costs limit the benefit of secondary indexes [18].

MV-PBT natively enables index-only visibility checks for additional access paths [18]. Different record types are augmented with timestamp information and a logical reference to its related version record in the main store in a space- and cache-efficient way. A more expensive regular visibility check and search succession in the MV-PBT main store is avoided by a lateral entry and traversal of the related partition as required.

Example Secondary Indexing MV-PBT. In Fig. 7, a MV-PBT storage management structure (MV-PBT Store) manages tuples with a key (*e.g.* '12') and several large attribute values (*e.g.* 'Maya; 20; 8;...'). A secondary index is maintained on attribute number 3 by an ordinary version-oblivious B$^+$-Tree and alternatively a version-aware MV-PBT index. Modifying transactions insert (TX_{U1}) and update (TX_{U2} and TX_{U3}) a logical tuple by the creation of version records (Regular Record (D), Replacement Records (C) and (B)). The B$^+$-Tree secondary index maintains index records (Y) and (Z), since attribute number 3 changes in (B). According to [18] and Section 3 (*Record Types in MV-PBT* and *Basic Operations*), modifying operations in a MV-PBT secondary index result in a Regular Record (5) and Replacement Record (4) in Partition 0 and another Replacement (3) and Anti Record (2) in Partition 1, due to possible interim partition switch process.

Reading transaction (TX_{R1},TX_{R2},TX_{R3}) started when modifying transactions (TX_{U1},TX_{U2},TX_{U3}) have respectively committed, hence snapshot related version records vary. Only TX_{R1} is expected to return (D) *{Maya;29}*, whilst result sets of other transactions are empty.

Version-oblivious B$^+$-Tree secondary indexes are traversed from (X) to (Y) or (X) to (Z), based on the search predicates. Either way, a reference to the logical tuple with the unique search key attribute value 12 is gathered and a regular search operation is performed in the MV-PBT Store. Traversal operations and visibility checks are successively performed for each partition and matching version record, until the search algorithm breaks. For transaction TX_{R1} 3 traversal operations from (A) to (B), (A) to (C) and (A) to (D) are necessary, however, only

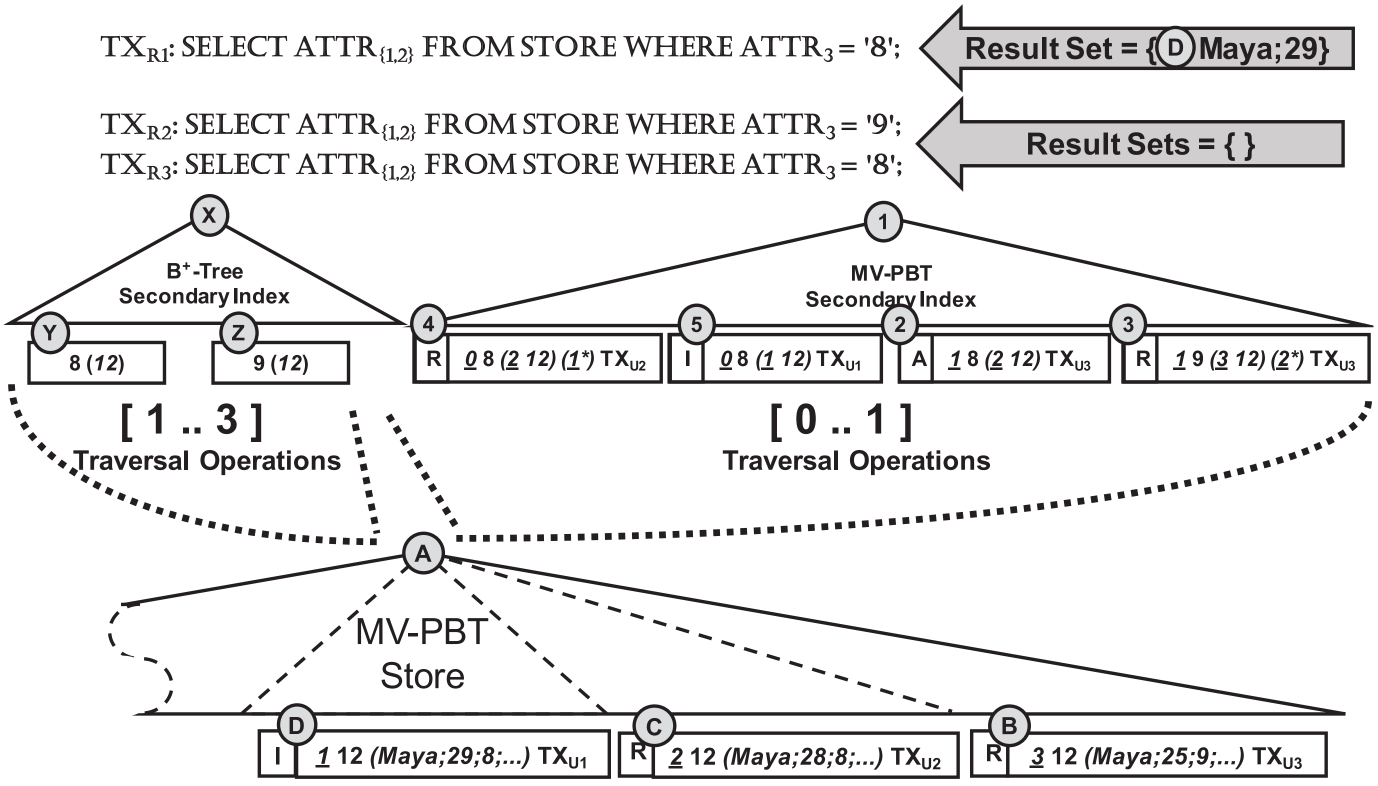

TXR1: SELECT ATTR{1,2} FROM STORE WHERE ATTR3 = '8';
Result Set = {D Maya;29}
TXR2: SELECT ATTR{1,2} FROM STORE WHERE ATTR3 = '9';
TXR3: SELECT ATTR{1,2} FROM STORE WHERE ATTR3 = '8';
Result Sets = { }
X
B+-Tree Secondary Index
Y 8 (12)
Z 9 (12)
[1 .. 3] Traversal Operations
1
MV-PBT Secondary Index
4 R 0 8 (2 12) (1*) TXU2
5 I 0 8 (1 12) TXU1
2 A 1 8 (2 12) TXU3
3 R 1 9 (3 12) (2*) TXU3
[0 .. 1] Traversal Operations
A
MV-PBT Store
D I 1 12 (Maya;29;8;...) TXU1
C R 2 12 (Maya;28;8;...) TXU2
B R 3 12 (Maya;25;9;...) TXU3

Fig. 7. Index-only visibility checks in secondary indices avoid unnecessary traversal operations in MV-PBT stores.

(D) is related to its snapshot and added to the result set. TX_{R2} breaks at (C) after 2 traversals and TX_{R3} at (B) after 1 traversal with an empty result respectively, due to mismatching search predicates. As a worst-case scenario, an insertion operation aborts and leave unrelated ghost records in the secondary index behind, but remove obsolete version records in the MV-PBT Store by garbage collection. As a result, every partition is traversed and searched.

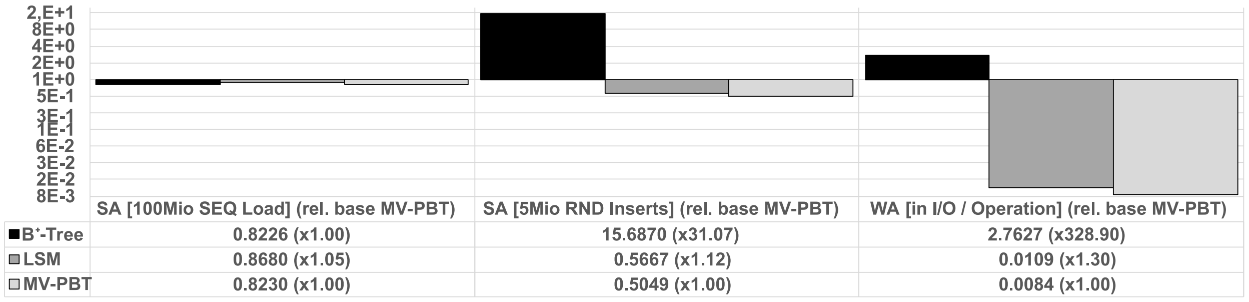
Version-aware MV-PBT secondary indexes perform index-only visibility checks. Therefore, only logical references of version records are returned (partition number and search key), which are related to the snapshot of the calling transaction. In the case of TX_{R1} (Fig. 7), the traversal begins from (1) to (2), however, the processed record is not related to the transaction snapshot. Hence, it is traversed from (1) to (4) (unrelated) and searched up to (5). The augmented transaction timestamp is related to the transaction snapshot and its reference was not invalidated by a visible index record. Hence, the MV-PBT Store is traversed from (A) to (D) by a lateral entry and the version record is added to the result set. For TX_{R2}, the secondary index is traversed from (1) to (3), which is unrelated to the transaction snapshot and no further record is found. Accordingly, for TX_{R3}, it is traversed from (1) to (2). The Anti Record is related to the transaction snapshot, whereby the visible Replacement Record (4) is invalidated and skipped. In both cases, the MV-PBT Store is not accessed. Moreover, potential ghost records in the MV-PBT secondary index are processed by the index-only visibility check and are unrelated to transaction snapshots. Selectivity is considered and typically more expensive operations in the main store are avoided.

The capability of inherent index-only visibility checking enable MV-PBT to serve as sole storage and index management structure with robust and constant access latencies, which are independent of the number of version records in a logical tuple's version chain [18].

7. Time-Travel Capabilities enabled by Multi-Version Storage

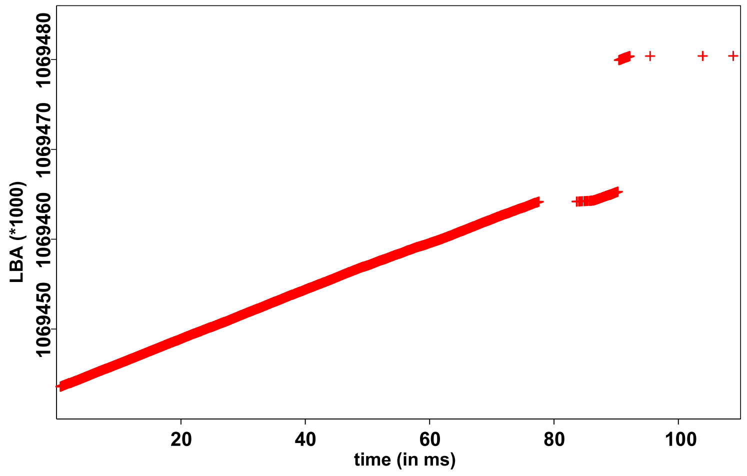
Modifications to the dataset in MV-PBT storage management structures yield insertions of new version records related to a logical tuple. Version records of different types are augmented with timestamp information for (in-) validation in order to perform an (index-only) visibility check in storage engines with MVCC and snapshot isolation. By this means, MV-PBT combines high parallelism of mutually non-blocking MVCC transaction management schemes with the benefits in append-based out-of-place storage and index management with low write-amplification. Considering recent trends in workload evolution, version chains of logical tuples tend to comprise multiple version records, which are relevant to different transaction snapshots of concurrently executing transactions. For instance, MV-PBT enables constant access latencies and robust performance in HTAP workloads [18] for arbitrary version records close as well as far of the entry point of a version chain.

Arbitrary Time-Travel Querying. MV-PBT inherently retains the entire history of each logical tuple, since modifications are maintained by the out-of-place insertion of a new version record. Garbage collection approaches are the only way to remove obsolete version records in the background, however, with time-travel these will never become obsolete for committed transaction timestamps. Time-travel queries on historical data perform robustly [18], since recently appended partitions are simply skipped by auxiliary filters, *i.e.* if a partition's minimum transaction timestamp logically succeeds a transaction snapshot, its comprised data cannot be related and is skipped by the search algorithm. Nevertheless, modifying workloads and scans potentially suffer from increasing fragmentation by caching and processing of intermingled unrelated version records in ancient partitions. Flexible strategies in creation of *Cached Partitions* might lessen these effects.



	SA [100Mio SEQ Load] (rel. base MV-PBT)	SA [5Mio RND Inserts] (rel. base MV-PBT)	WA [in I/O / Operation] (rel. base MV-PBT)
■ B+-Tree	0.8226 (x1.00)	15.6870 (x31.07)	2.7627 (x328.90)
▨ LSM	0.8680 (x1.05)	0.5667 (x1.12)	0.0109 (x1.30)
□ MV-PBT	0.8230 (x1.00)	0.5049 (x1.00)	0.0084 (x1.00)

(a) Relative space and write-amplification



(b) MV-PBT strictly fulfills sequential write patterns.

Fig. 8. Experiments 1 and 2 evaluate the structural properties of MV-PBT.

Application of Named Snapshots. In enterprise applications, it is mostly sufficient to provide specific points in historical data, *e.g.* in a daily or monthly reporting interval. User-defined *Named Snapshots* [2] are a possibility to sustain a consistent searchable historical snapshot of the dataset. Intermediate version records become obsolete and get removed by GC, whenever they are unrelated to any active transaction snapshot or maintained *Named Snapshot*. By this means, the number of unrelated version records is reduced and overall transactional throughput is increased. Moreover, MV-PBT could be instrumented to create optimized *Cached Partitions* or perform GC, based on the fragmentation and necessary write-amplification.

Out-of-place maintenance of timestamped version records in partitions makes MV-PBT a leading edge compared to its competitors in time-travel query processing. Whilst B$^+$-Tree is limited by intermingled version records and massive write-amplification, LSM-Tree additionally suffers from limited *Named Snapshot* support, due to necessary complex merge policies.

8. Experimental Evaluation

We present the analysis of MV-PBT as storage management structure in comparison beside the baselines LSM-Trees and B$^+$-Trees fully integrated in WiredTiger 10.0.1 (WT) [2]. LSM-Trees in WT build upon components of the provided B$^+$-Trees upon which MV-PBT is also implemented. A good comparability is achieved, since all structures commonly operate on equal code lines and B$^+$-Tree techniques, *e.g.*: prefix truncation, suffix truncation and snappy compression; reduced maintenance effort due to flexible page sizes; main memory page representation with sorted areas, update lists and insertion skiplists; MVCC transaction timestamps in main memory record representation; tree-based buffer management.

Experimental Setup. We deployed WiredTiger(WT) 10.0.1 and WT with MV-PBT as storage management structure on an *Ubuntu 16.04.4 LTS* server[1] with an eight core *Intel(R) Xeon(R) E5-1620* CPU, 2 GB RAM and an *Intel DC P3600* enterprise SSD. Applied parameter configurations constitute an economic setup for modern workload properties, *i.e.* huge dataset sizes with comparably low main memory volumes lessen caching effectiveness and generally yield disk-based access patterns (compare [33,34]). We used the YCSB framework [35,36] for experimental evaluation with a dataset size of approx. 50 GB, unless stated otherwise. The WT cache size is set to 100 MB and LSM-chunks as well as partitions are allowed to grow up to 20 MB in order to provide comparable results and enabling sufficient general cache for frequently accessed and traversed inner B$^+$-Tree nodes. Direct IO is enabled and the OS page cache is cleaned every second in order to ensure repeatable, reliable and even conservative results.

Experiment 1: Space and Write-Amplification. In Fig. 8(a), B$^+$-Tree, LSM-Tree (merges are disabled for comparability) and MV-PBT are initially bulk loaded with 100 million records (key and value size are 13 and 16 bytes respectively). Prefix truncation in record keys, suffix truncation in separator keys and snappy compression allow comparable relative space requirements for all approaches. There is a clear evidence of the synergy between prefix truncation and partitioned key, since the enlarged record key by a 2 byte partition number does not result in higher space requirements. Subsequently, 5 million new records are inserted — yielding approx. 60 new partitions/LSM-components. Due

[1] It is a matured version of Ubuntu to the beginning of our research. The operable system allows comparable results in a homogeneous environment (*e.g.* [17,18,27,28]).

to compression techniques, the additional relative space requirement is lower than the actually added record size, with slight advantages for MV-PBT. B$^+$-Tree suffer from insertions in the read-optimized layout due to node splits — yielding massive relative space-amplification per newly inserted records.

Insight: MV-PBT offers the lowest space-amplification, that is between 12% and 31× better. Finally, the write-amplification (Fig. 8(a)) is evaluated after 5 million inserts. Since almost each insertion causes escalating node splits in the read-optimized layout of a B$^+$-Tree, each insertion causes 2.76 write I/Os of half filled nodes. Sequential writes of dense-packed nodes allow LSM-Trees and MV-PBT to achieve singular writes of optimally filled nodes, yielding much less write I/O per insertion. MV-PBT achieves a better factor due to commonly used inner nodes. Moreover, merge operations of LSM components would cause a downturn of write-amplification by orders of magnitude.

Insight: Compared to LSM-trees, MV-PBT offers 30% less write-amplification and is up to 300× better than B-Trees.

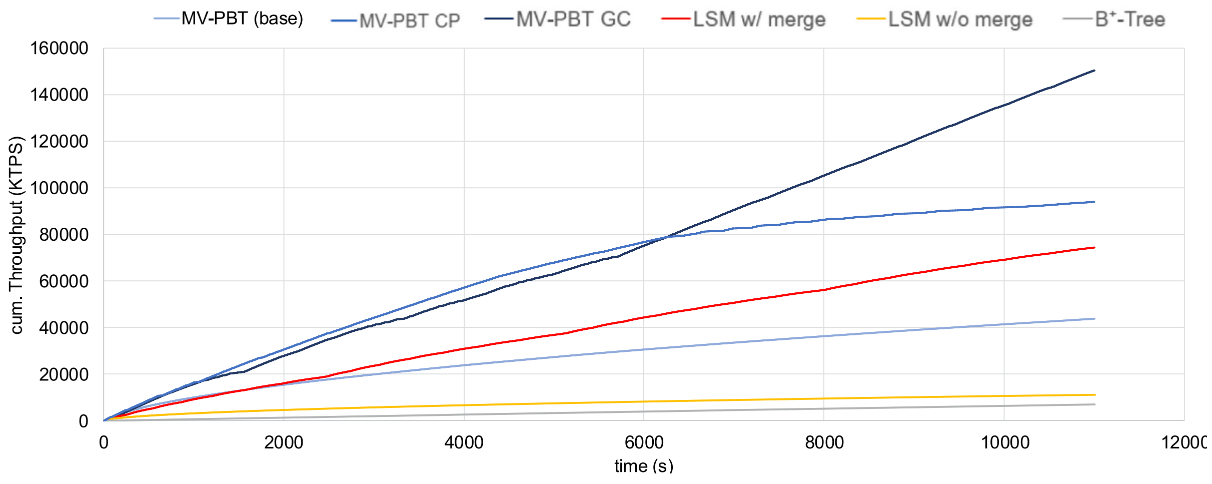
Experiment 2: Sequential Write Pattern. Evictions of *dirty buffers* from main memory cause write I/O operations to secondary storage devices, which are preferably performed in a sequential pattern, especially with Flash-SSDs, to achieve better performance and longevity. *Logical Block Addressing (LBA)* enables a logical sequential arrangement of physical storage units in SSDs, hence write I/Os to a sequence of LBAs over time indicate a beneficial sequential write pattern. Fig. 8(b) depicts the desired write pattern for MV-PBT with LBAs on the ordinate and evolving time on the abscissa. As a result of the partition switch operation, delayed maintenance operations (splits) on leaves followed by inner nodes are performed in a reconciliation operation. Afterwards, leaves are identified by a tree walk and ascendingly written to secondary storage devices, depicted by the continuously ascending markers. Finally, the referencing levels of immutable inner nodes are sequentially written, depicted by multiple shorter continuously ascending markers.

Insight: MV-PBT is able to perform advantageous sequential writes.

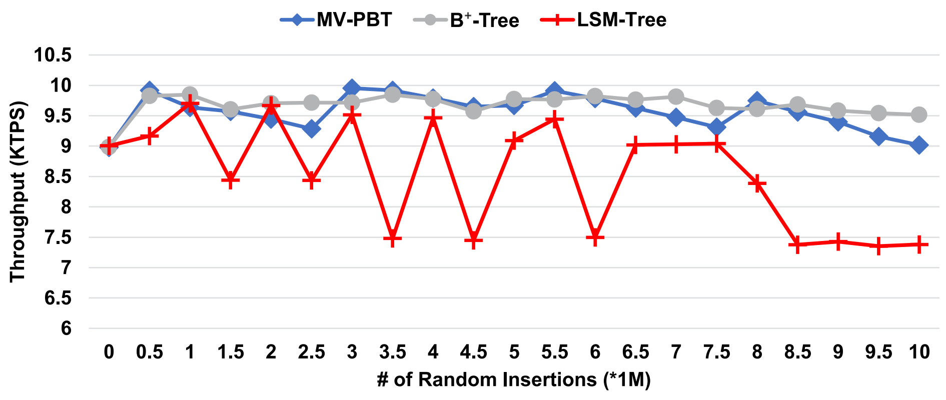
Experiment 3: Steady Performance by Cached Partitions and Garbage Collection. The write-heavy YCSB Workload A consists of 50% updates and reads, respectively (depicted in Fig. 9(a)). Write-amplification in B$^+$-Trees yield poor performance characteristics (7M tx). Sequential writes and low write-amplification in base MV-PBT (no Cached Partition and GC) allow much higher transactional throughput, however, increasing search effort degenerates performance (44M tx), whereby LSM-Trees hold search effort down by merges (74M tx).

Insight: The direct structural comparison of LSM-Trees and MV-PBT is without merges and garbage collection, whereby MV-PBT outperforms LSM (11M tx) by 4×. Enabling Cached Partitions allow MV-PBT increased read efficiency, however, memory footprint of auxiliary filter structures degenerates its capabilities over time due to effectively reduced cache (94M tx).

Insight: Occasional Garbage Collection in MV-PBT (every 400 Partitions) enables stable performance characteristics (151M tx), outperforming LSM-Trees by 2×.



(a) Accumulated executed transactions (*1k) in YCSB Workload A with 1kB value size



(b) Intermediate State Read-Only Throughput

Fig. 9. Experiments 3 and 4 evaluate consistent performance of MV-PBT.

Experiment 4: Read-Only Performance Characteristics of intermediate Structures States. YCSB Workload C is performed several times after inserting 500k small random records for 10 min, respectively (depicted in Fig. 9(b)). B$^+$-Tree remain very stable, but slightly decrease, since the read-optimized layout breaks. LSM-Trees throughput is varying based on the number of LSM components.

Insight: Commonly cached inner nodes and periodically created Cached Partitions allow MV-PBT to retain comparable read performance even if 80 partitions are created after 10 million random insertions.

Experiment 5: Impact of Different Value Sizes. YCSB basic workloads (Fig. 10) are performed on small (16 bytes), medium (100 bytes) and large (1000 bytes) value sizes, the initial load has been adjusted to match approx. 50 GB dataset size.

Insight: MV-PBT outperforms its competitors in the high and medium update intensive workloads A and B, even the LSM-Tree by 2× in the workload A. The read-only workload C is performed on the read-optimized layout after load phase — comparable results prove negligible costs of partitioned key comparisons, whereas LSM-Trees are only able to retain performance for one component (Figs. 10(c) and 9(b)). Workload D searches for few concurrently inserted records. B$^+$-Tree benefits from well cached nodes in the traversal path due to the recent insertion. This is also valid for MV-PBT and LSM-Trees, however, concurrent insertions are not in the MVCC snapshot and cause search operations in other partitions or components, which is 2× faster in MV-PBT. Finally, MV-PBT is able to achieve comparable performance

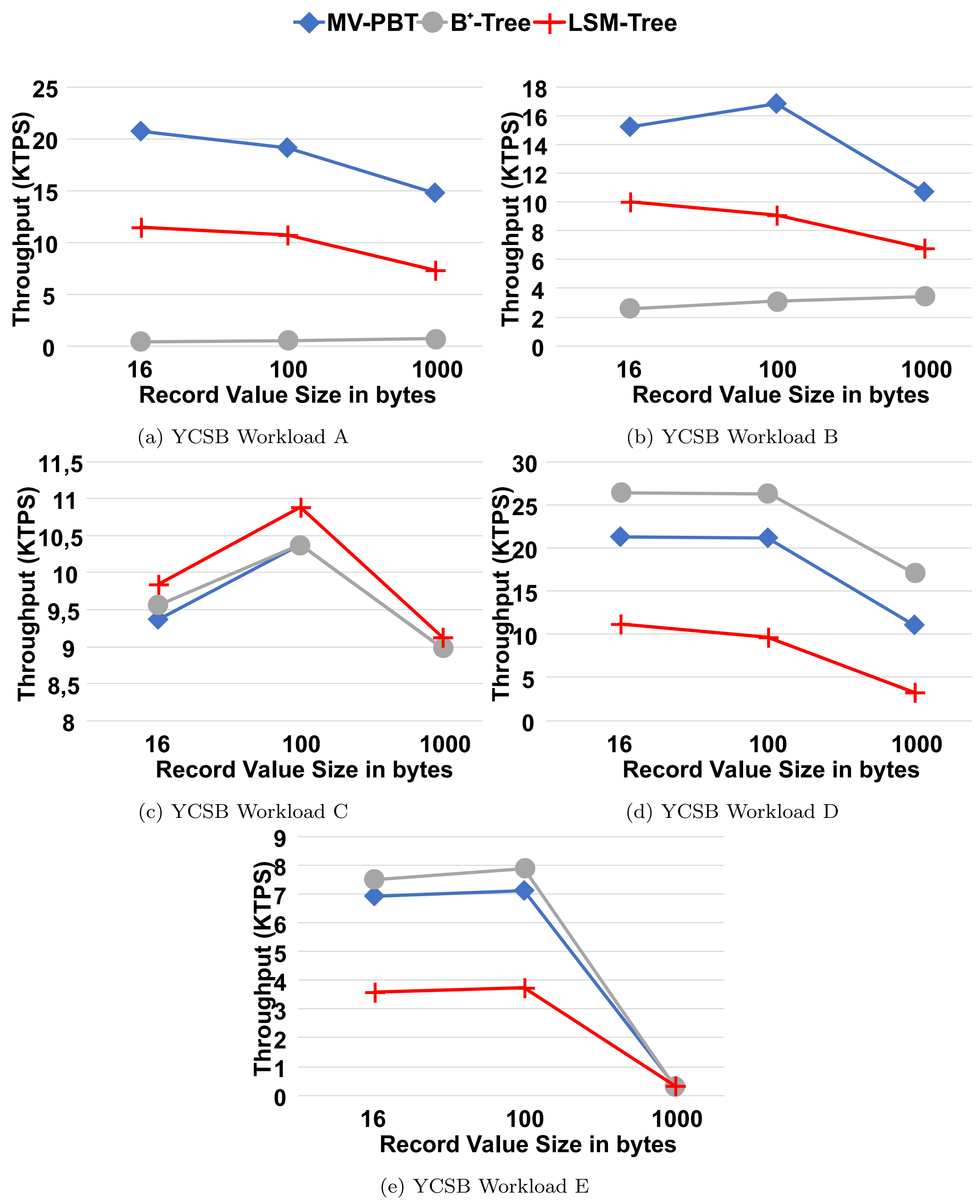


Fig. 10. Experiment 5 evaluates performance for different value sizes.

to B$^+$-Tree in the mostly scan workload E. Cached Partitions and commonly cached inner nodes enable cheap merge sort scan operations.

Experimental Setup for subsequent Time-Travel Experiments 6 and 7. The K/V-Store is initially loaded with approx. 120GB of 1KB key/value pairs and a Named Snapshot is created. The WT cache size is set to 200 MB (including 40 MB partition/chunk buffer). YCSB Workload E (95% scan, 5% insert) is performed in order to simulate background I/O and occupying bandwidth. $10k/100k$ (0.01%/0.1%) of initially loaded tuples are randomly selected and updated step-wise (52× in total). Meanwhile, whenever every second update process completes, the update stream is paused and time-travel queries are performed on the Named Snapshot for all initially selected tuples and the total latency is measured.

Experiment 6: Robust Latencies in MV-PBT on History Data. In Fig. 11, time-travel query latencies ($10k$ single operations) are depicted for different version chain lengths. MV-PBT (w/o timestamp (TS) filtering per partition)'s (brighter blue square) latency successively downgrades, as version chain length increase by inserted version records with updated values. Modifying operations increase fragmentation, since up to 20 partitions are created over time, in order to absorb updates. The search algorithm proceeds and traverses every partition from the most recent to the lowest numbered one. Neither applied bloom filters on search key attribute values nor Cached Partitions prevent partitions from being accessed. However, these partitions do not contain any version record, which is related to the time-travel query.

A second effect can be identified. Modifying workloads leave messy page contents behind for leaves as well as inner nodes due to modern lock-free B$^+$-Tree techniques, *e.g.* update lists or insert skiplists. Time-travel queries are harmed by current workload. WT restores a read-optimized disk layout on a reconciliation process, hence partition switch processes incidentally improve read performance, what is imposingly demonstrated in excerpts at 2, 6, 24, 38 or 48 successor versions.

MV-PBT (min. TX TS) (darker blue diamond in Fig. 11) overcomes these issues by space-efficient auxiliary filter structures on transaction timestamps. One *minimum transaction timestamp* per partition is sufficient due to MV-PBT's append-based horizontal partition management. A time-travel query is performed on a valid transaction snapshot (in this case an initially created snapshot), hence it is sufficient to search and traverse partitions that already existed at the transaction snapshot. Subsequently created higher numbered partitions are skipped, whenever the partition's minimum transaction timestamp logically succeeds the transaction snapshot. Moreover, minimum transaction timestamps effectively avoid interferences with modifying workloads, whenever commonly used inner nodes sustain read-optimized disk layout (>= 10 successor versions). *As a result, MV-PBT exhibits robust and up to 35% lower latencies in time-travel query processing.*

Experiment 7: Time-Travel Capabilities in Storage Management Structures. In Fig. 12, time-travel query latencies are depicted for different number of updates (X-axis) and modified data share (Figs. 12(a) and 12(b)). LSM-Trees as well as B$^+$-Trees remove intermediate version records by different garbage collection techniques. Moreover, LSM-Trees do not support *Named Snapshots*, whereas it is instructed to return the lowest timestamped and available version record. By this means, time-travel capabilities are simulated, even though returned version records and version chain lengths vary. *Only MV-PBT maintains every version record in this configuration, since version support and performance of others is limited.*

B$^+$-Trees intermingle modifying workload and history snapshot data, since they maintain a strict lexicographical sort order. The version chain length is at most 3 and modifications are paused, however, the read-optimized layout is broken and the secondary storage device's bandwidth is occupied by insertions and scans of YCSB Workload E in the background. B$^+$-Trees perform an order of magnitude worse than their competitors and are not suitable for time-travel query processing.

LSM-Tree's latencies vary depending on the number of version records according to the number of LSM components. Meanwhile, LSM-Trees are able to minimize searchable components in a read-optimized layout. Moreover, insertions of YCSB Workload E in the background are well buffered in the main memory component. Nevertheless, occasionally occurring better performance compared to MV-PBT is explained with the lack of appropriate time-travel capabilities. That is, at these points the LSM-trees have less than the specified number of versions due to background compactions.

MV-PBT exhibits robust time-travel query latencies. Insertions of YCSB workload E are well absorbed by the most recent partition, whilst minimum transaction timestamps protect appended partitions. Moreover, every intermediate version record is contained, until comprising partitions are explicitly dropped by user-intended garbage collection processes, *i.e.* MV-PBT retains the entire tuple history. Arbitrary transaction snapshots are available for query processing. Finally, time-travel query processing latencies are reduced by 20% in average, compared to varying LSM-Trees.

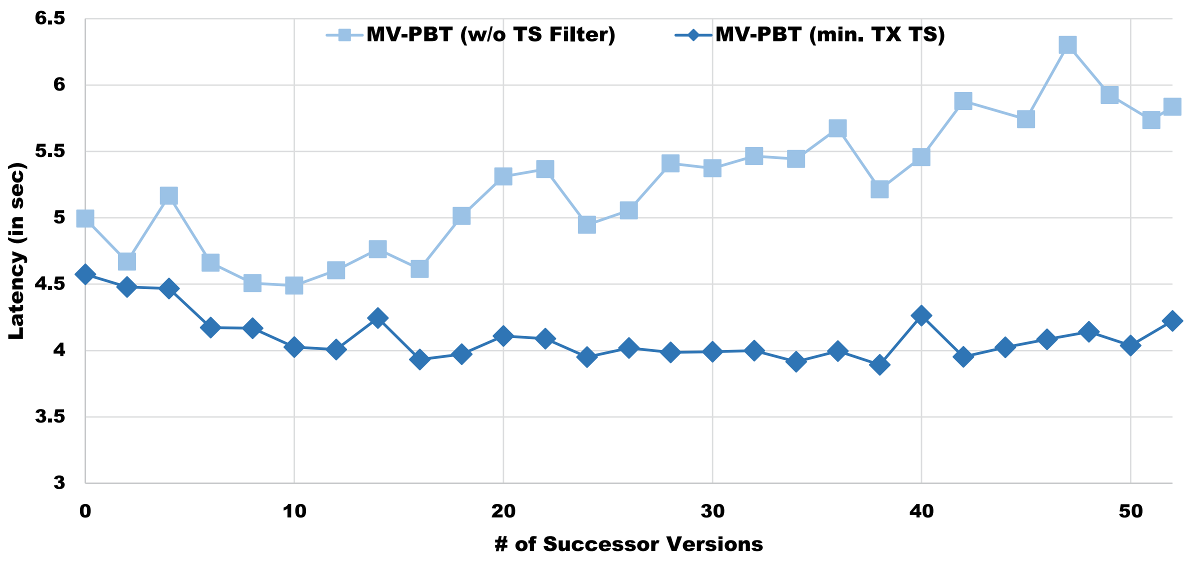
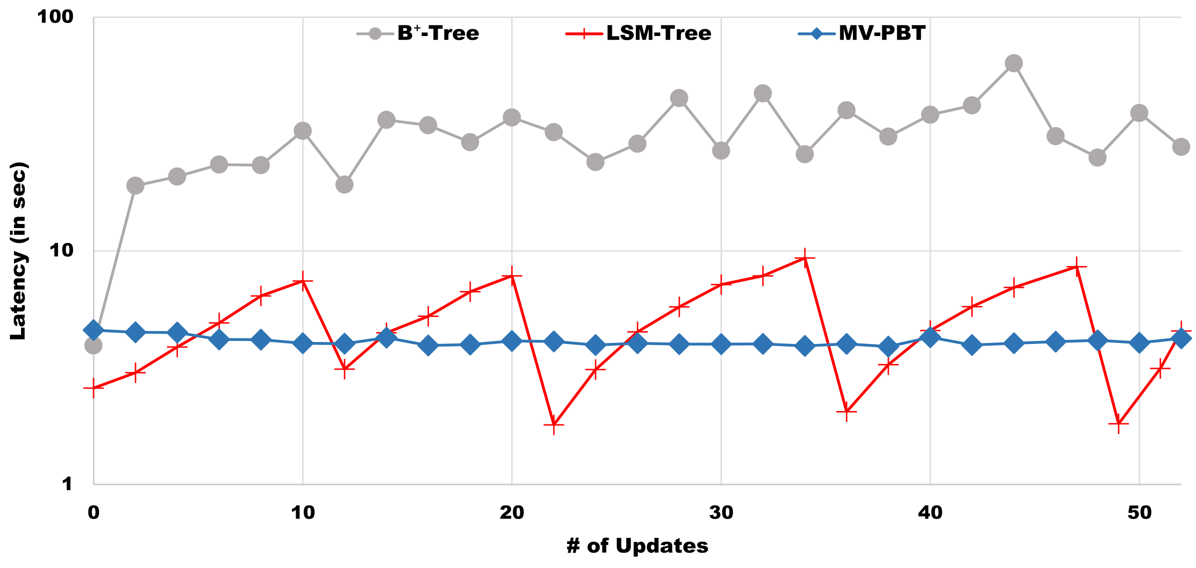
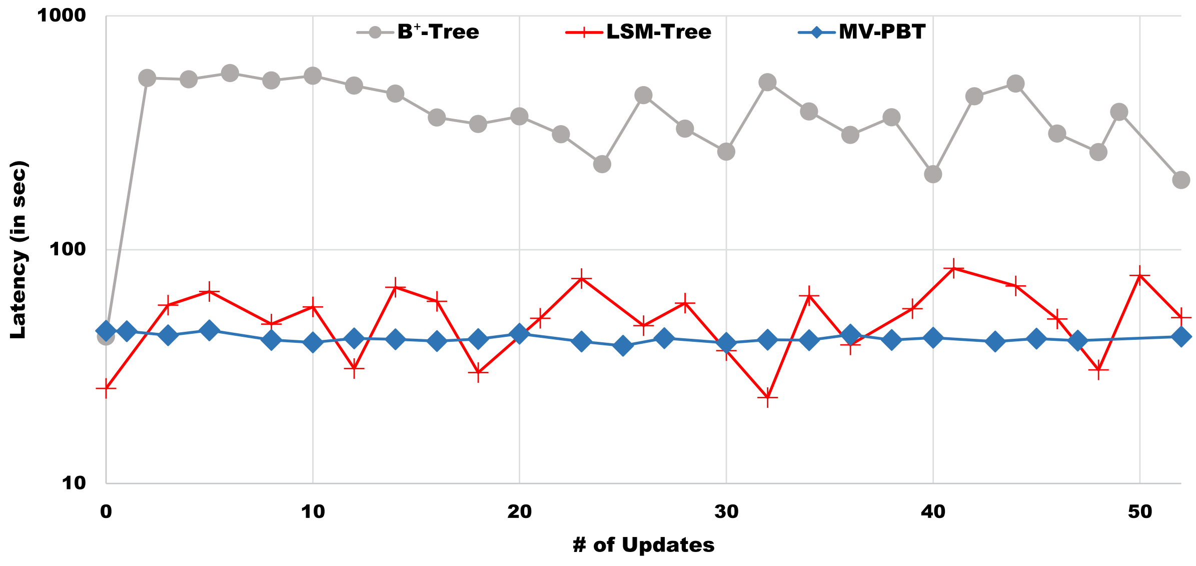


Fig. 11. MV-PBT exhibits robust Time-Travel Query Performance, due to minimum timestamp filtering per MV-PBT partition (min. TX TS).



(a) Result set size $10k$



(b) Result set size $100k$

Fig. 12. Only MV-PBT is capable to comprise entire Version History.

9. Conclusion

In this paper we present Multi-Version Partitioned BTrees (MV-PBT) as a sole storage and index management structure [18] in KV-storage engines. Logical horizontal partitioning yields beneficial appends of version records within a single tree structure. Partitions leverage properties of B$^+$-Trees by common utilization and caching of inner nodes in traversal operations, whereby constant search performance and high fragmentation are brought together. This behavior leveraged by Cached Partition in order to minimize write-amplification to secondary storage devices. Contrary to LSM-Trees, merging is considered for garbage collection of obsolete version records instead of sustained search performance. Therefore, MV-PBT enables robust latencies in arbitrary time-travel query processing. Finally, (index-only) visibility checks enable reliable result sets in appropriate storage and index management structures without the need of expensive downstream visibility checks, wherefore MV-PBT is predestinated to be applied in KV-storage engines.

CRediT authorship contribution statement

Christian Riegger: Conceptualization, Formal analysis, Investigation, Methodology, Resources, Software, Validation, Visualization, Writing – original draft, Writing – review & editing. **Ilia Petrov:** Conceptualization, Formal analysis, Methodology, Project administration, Resources, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] Facebook, RocksDB A persistent key-value store, 2022, URL http://rocksdb.org.
[2] MongoDB-Inc., WiredTiger: WiredTiger Developer Site, 2021, URL https://source.wiredtiger.com/.
[3] J.J. Levandoski, D.B. Lomet, S. Sengupta, The Bw-Tree: A B-tree for new hardware platforms, in: ICDE, 2013.
[4] MongoDB, MongoDB: The Application Data Platform, 2022, URL https://www.mongodb.com.
[5] Facebook, MyRocks A RocksDB storage engine with MySQL, 2022, URL http://myrocks.io.
[6] R. Bayer, E. McCreight, Organization and Maintenance of Large Ordered Indices, SIGFIDET '70, New York, NY, USA, 1970.
[7] P.E. O'Neil, E. Cheng, D. Gawlick, E.J. O'Neil, The Log-Structured Merge-Tree (LSM-Tree), Acta Inf. (1996).
[8] R. Sears, R. Ramakrishnan, BLSM: A General Purpose Log Structured Merge Tree, in: SIGMOD, 2012.
[9] R. Bayer, K. Unterauer, Prefix B-trees, ACM Trans. Database Syst. (1977).
[10] X. Ruan, Z. Zong, M. Alghamdi, Y. Tian, X. Jiang, X. Qin, Improving write performance by enhancing internal parallelism of Solid State Drives, in: IPCCC, 2012.
[11] Y.A. Winata, S. Kim, I. Shin, Enhancing internal parallelism of solid-state drives while balancing write loads across dies, Electron. Lett. (2015).
[12] I. Shin, Verification of performance improvement of multi-plane operation in SSDs, Int. J. Appl. Eng. Res. (2017).
[13] D. Ma, J. Feng, G. Li, A Survey of Address Translation Technologies for Flash Memories, ACM Comput. Surv. (2014).
[14] F. Chen, D.A. Koufaty, X. Zhang, Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives, in: SIGMETRICS, 2009.
[15] R. Gottstein, Impact of New Storage Technologies on an OLTP DBMS, its Architecture and Algorithms (Ph.D. thesis), TU, Darmstadt, 2016.
[16] S. Idreos, M. Callaghan, Key-Value Storage Engines, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 2667–2672, http://dx.doi.org/10.1145/3318464.3383133.
[17] C. Riegger, I. Petrov, Storage Management with Multi-Version Partitioned BTrees, in: Advances in Databases and Information Systems: 26th European Conference, ADBIS 2022, Turin, Italy, September 5–8, 2022, Proceedings, Springer-Verlag, Berlin, Heidelberg, 2022, pp. 255–269, http://dx.doi.org/10.1007/978-3-031-15740-0_19.
[18] C. Riegger, T. Vinçon, R. Gottstein, I. Petrov, MV-PBT: Multi-Version Index for Large Datasets and HTAP Workloads, in: EDBT, 2020.
[19] C. Luo, M.J. Carey, LSM-based storage techniques: A survey, VLDB J. 29 (1) (2020) 393–418.
[20] N. Dayan, S. Idreos, The Log-Structured Merge-Bush & the Wacky Continuum, in: Proc. SIGMOD, 2019, pp. 449–466.
[21] N. Dayan, S. Idreos, Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging, in: Proc. SIGMOD, 2018, pp. 505–520.
[22] H. Zhang, H. Lim, V. Leis, D.G. Andersen, M. Kaminsky, K. Keeton, A. Pavlo, Surf: Practical Range Query Filtering with Fast Succinct Tries, SIGMOD '18, 2018.
[23] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, S. Idreos, Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores, in: SIGMOD, 2020.
[24] C. Riegger, A. Bernhardt, B. Moessner, I. Petrov, bloomRF: On Performing Range-Queries with Bloom-Filters based on Piecewise-Monotone Hash Functions and Dyadic Trace-Trees, 2020, CoRR.
[25] W. Zhong, C. Chen, X. Wu, S. Jiang, REMIX: Efficient Range Query for LSM-trees, in: FAST, 2021.
[26] G. Graefe, Sorting And Indexing With Partitioned B-Trees, 2002.
[27] C. Riegger, T. Vinçon, I. Petrov, Write-Optimized Indexing with Partitioned B-Trees, in: iiWAS, 2017.
[28] C. Riegger, T. Vinçon, I. Petrov, Indexing Large Updatable Datasets in Multi-Version Database Management Systems, in: IDEAS, 2019.
[29] C. Riegger, T. Vinçon, I. Petrov, Multi-Version Indexing and Modern Hardware Technologies: A Survey of Present Indexing Approaches, in: Proceedings of the 19th International Conference on Information Integration and Web-Based Applications & Services, iiWAS '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 266–275, http://dx.doi.org/10.1145/3151759.3151779.
[30] R. Gottstein, I. Petrov, S. Hardock, A.P. Buchmann, SIAS-Chains: Snapshot Isolation Append Storage Chains, in: R. Bordawekar, T. Lahiri (Eds.), in: ADMS@VLDB, 2017.
[31] Y. Matsunobu, S. Dong, H. Lee, Myrocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph, Proc. VLDB Endow. 13 (12) (2020) 3217–3230, http://dx.doi.org/10.14778/3415478.3415546.
[32] MonogDB, RocksDB Storage Engine Module for MongoDB, 2023, URL https://github.com/mongodb-partners/mongo-rocks.
[33] D. Lomet, Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed, in: Proceedings of the 14th International Workshop on Data Management on New Hardware, DAMON '18, Association for Computing Machinery, New York, NY, USA, 2018, http://dx.doi.org/10.1145/3211922.3211927.
[34] T. Neumann, M.J. Freitag, Umbra: A Disk-Based System with In-Memory Performance, in: 10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, the Netherlands, January 12-15, 2020, Online Proceedings, www.cidrdb.org, 2020, URL http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf.
[35] J. Ren, C. Kjellqvist, L. Deng, GitHub - basicthinker/YCSB-C: Yahoo! Cloud Serving Benchmark in C++, 2021, URL https://github.com/basicthinker/YCSB-C.
[36] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking Cloud Serving Systems with YCSB, SoCC '10, 2010.