Multi-Version Indexing and modern Hardware Technologies

A Survey of present Indexing Approaches

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

Tobias Vinçon
Data Management Lab
Reutlingen University, Germany
DXC Technology
Böblingen, Germany
tobias.vincon@dxc.com

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

ABSTRACT

Characteristics of modern computing and storage technologies fundamentally differ from traditional hardware. There is a need to optimally leverage their performance, endurance and energy consumption characteristics. Therefore, existing architectures and algorithms in modern high performance database management systems have to be redesigned and advanced. Multi Version Concurrency Control (MVCC) approaches in data-base management systems maintain multiple physically independent tuple versions. Snapshot isolation approaches enable high parallelism and concurrency in workloads with almost serializable consistency level. Modern hardware technologies benefit from multi-version approaches. Indexing multi-version data on modern hardware is still an open research area. In this paper, we provide a survey of popular multi-version indexing approaches and an extended scope of high performance single-version approaches. An optimal multi-version index structure brings look-up efficiency of tuple versions, which are visible to transactions, and effort on index maintenance in balance for different workloads on modern hardware technologies.

CCS Concepts

•Information systems → Data access methods;

Keywords

Index Structures, MVCC, Modern Hardware Technologies

1. INTRODUCTION

Nowadays, Multi-Version Database Management Systems (MV-DBMS) are widely spread and commonly used in commercial and academia. There are many reasons for applications storing data in a multi-version manner to keep a history for temporal querying, e.g. in financial applications or software development. Further reasons are based on providing more parallelism in reading and writing transactions without losing consistency of data in multi version concurrency control (MVCC) and snapshot isolation (SI).

Trends in modern computing and storage technologies are receiving growing attention within the database research community. Processing data in modern multi-core processors (CPU) and graphics processors (GPU) bring benefits in parallel computing. Furthermore, power efficient and flexible configurable field programmable gate arrays (FPGA) allow fast processing of huge amounts of data near its storage location and reduce data movement along memory hierarchies. Flash-based storage media like Flash Solid State Disks (SSD) as well as upcoming non-volatile memories (NV-RAM) become state of the art in modern memory hierarchies. The characteristics of modern storage media differ from traditional storage media. Read/Write asymmetry, low latencies, parallelism, out-of-place updates, erases and wear are the characteristics to address in MV-DBMS algorithms. There are benefits in writing once sequentially and small random reads in parallel are almost as fast as sequential reads. Furthermore, new storage media coexist with traditional symmetric storage media in complex memory hierarchies.

Characteristics of modern hardware match well to multi-version data and enterprise workloads. An update results in a creation of a new version. Old versions are retained immutably, so several versions of a data tuple exist. Versions are stored and processed as independent entities. For the extra effort of a visibility check, more parallelism is achieved, because reading transactions are never blocked by writing transactions. CPUs, GPUs and FPGAs are kept busy and do not idle. In theory, flash benefits from independent immutable tuple versions. Updates create a new version out-of-place, so older versions are never changed and have not to be erased and rewritten on flash, except for garbage collection. As a result, write amplification, erases, wear and energy consumption are reduced. Several versions can be read in parallel due to fast parallel reads. [2] optimized multi-version data placement and invalidation problems, so that creating new versions result in pure append-only writes and characteristics of flash are optimally leveraged for multi-version data in MV-DBMS base tables.

Indexing multi-version data is still an open research area, considering characteristics of modern hardware. In general, MV-DBMS use the ubiquitous B^{+}-Tree [8] for indexing data

iiWAS '17, December 4–6, 2017, Salzburg, Austria

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-5299-4/17/12...$15.00

DOI: https://doi.org/10.1145/3151759.3151779

in a sorted data structure. In MV-DBMS and modern hardware, this approach brings some drawbacks. *First*, the index structure has to avoid false negatives, for which reason every version of a tuple is maintained as an index record. So, the dataset of a B$^+$-Tree in a MV-DBMS is considerably larger than in a single-version approach. Based on the version chain ordering of the DBMS, the amount of required index records can be reduced. *Second*, newly created tuple versions on update are applied to the index as new independent records. On eviction of index nodes from buffer cache, it results in several random write I/O and high write amplification on persistent storage media, like SSDs. *Third*, in MVCC snapshots, maintained data in indexes is not sufficient for returning the tuple version, that is visible to a transaction, even if every search predicate is an indexed column, because transaction timestamps for visibility check are exclusively maintained at tuple versions in base tables. The index returns a set of candidates, which have to be rechecked for visibility in base tables. So, more random read I/O on base tables occur. A schematic representation of a visibility check is depicted in Figure 1. *Last*, several index records are not visible to a transaction, but have to be processed in an index scan, too. Garbage collection (GC) cycles reduce number of index records, that are no more visible to any active transaction, but can also result in high write amplification and a random write I/O pattern.

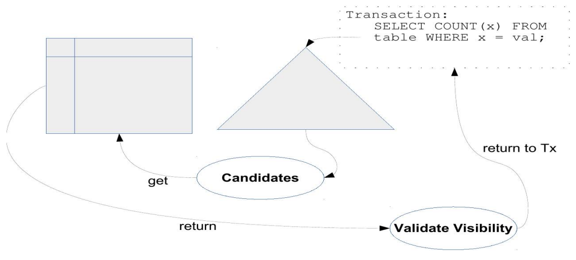

Figure 1: Visibility Check

In this paper, we define required index structure characteristics in a MV-DBMS for modern hardware technologies in Section 2. We give a survey of four popular multi-version and temporal indexing approaches in Section 3, whereby we give a detailed description of our findings. Furthermore, we provide an overview of indexing approaches, which can handle characteristics of modern hardware very well, but have no multi-version capabilities in an extended scope in Section 4. Finally, we propose one indexing approach for further research in Section 5, which can be optimized for requirements mentioned in Section 2.

2. REQUIRED INDEX STRUCTURE CHARACTERISTICS

Nowadays, a multi-version index structure has to fulfill following characteristics. *First*, writes on secondary storage media should be performed sequentially and append-only, if possible. Although, random read I/O can be approved due to high parallelism and fast reads in asymmetric storage media. *Second*, in-place updates must be minimized, because of asymmetry, out-of-place updates and wear in flash. Invalidation at index records, data modification, maintenance operations and garbage collection all over an index structure have the result of logical in-place updates of whole pages on secondary storage. Nonetheless, redundancy and index size should be kept as small as possible. *Third*, bulk loads are very common operations in modern DBMS, e.g. in case of bulk inserts, index creation or replication. In traditional indexing approaches, like B$^+$-Trees, sometimes it is cheaper to rebuild the whole index from scratch, rather than loading huge amounts of data to an existing index structure. Indexes should be able to handle bulk loads with a sequential write I/O pattern and low maintenance effort. *Fourth*, maintained data in multi-version index structures has to be sufficient for visibility check, to eliminate unnecessary reads on base tables, especially in case of row-oriented DBMS. Therefore, frequently queried tuple attributes as well as validation and invalidation timestamps have to be applied to index records. Furthermore, natural order of predecessor and successor versions must be considered. *Last*, workload adaptivity is required. In complex memory hierarchies, data placement is fundamental for fast data access and do not let cores in processing units idle. Generally, not all data fits in fast main memory and must be fetched from further storage media. Based on current workload, data has to be partitioned in hot and cold data and located on different devices with different characteristics in memory hierarchy. A detailed overview of an optimal index structure for modern hardware technologies and multi-version data is listed in Table 1.

Table 1: Characteristics of a near-optimal Index Structure

Characteristic	Optimal
Secondary Storage I/O Pattern	
Write Pattern	Sequential append-only write I/O
Read Pattern	Latencies of random reads are near to sequential reads due to high parallelism in flash, optimally required data is located as close as possible to the processing unit
Index Structure and Operations	
Insert Operations	Inserts only in main memory can be performed very fast and guarantee minimal write amplification
Update/Invalidation Operations	Out-of-place updates in main memory can be performed fast and guarantee minimal write amplification
Bulk Loads	Inserts of huge amounts of records only in main memory can be performed very fast and guarantee minimal write amplification
Maintenance Operations	Maintenance slows down insert performance, but is a mandatory operation for good look-up performance – performing as much maintenance as possible in main memory, without evicting index pages, will reduce write amplification and wear
Garbage Collection	As soon as possible for records of tuple versions, that are not visible to any active transaction, optimal is in main memory, if records are already flushed to secondary storage, try to keep sequential write I/O pattern up
Redundancy	The less, the better – considering out-of-place invalidation, maximal two records for the live span of one tuple version
Index Size	The less, the better – for performing visibility checks only in index structure, a key, (in-)validation timestamps and record id are required
Multi-Version Capabilities	
Visibility Check	Maintained data in index structure must be sufficient to perform visibility checks
Version Chain Ordering	Successor and predecessor versions should be well connected, newer versions are likely requested and should be kept in main memory rather than records of older tuple versions
Workload Adaptivity	
Workload Adaptivity	Index structure should have the capability to optimize data placement along memory hierarchy for switches in workload

3. CURRENT MULTI-VERSION INDEXING APPROACHES

In this section, we present four popular multi-version and temporal indexing approaches. We give a short overview of their structure and how index operations are performed. Furthermore, we investigate their capabilities mentioned in Section 2 and sum up our findings in short review tables. We will show that every indexing approach has shortcomings in at least one of the areas of multi-version capabilities, complexity in look-up and maintenance, alignment to modern computing and storage technologies and their specialization for single fields of application.

3.1 Time-Split B-Tree

3.1.1 *Data Structure Overview*

A Time-Split B-Tree [9], depicted in Figure 2, is a two-dimensional tree-based index structure with the dimensions key value and timestamp, which enables efficient querying of temporal data. An index record consists of key columns, a validation timestamp and the record id of the tuple version (see Figure 3). An invalidation of a version is performed out-of-place by inserting a new index record with same key value, a timestamp and a new record id. Therefore, key columns have to be unique. Several tuple versions are held on a page sorted by the key dimension. If a page is getting filled and there is insufficient space to insert an additional record on the target page, there are two possibilities to split a page. Several policies exist for choosing in which dimension a split will be performed. A Time-Split separates historical from current data. The separation in historical and current data depends on a timestamp determined at Version-Split. It can be the split time or any point in time before the split time, but after the oldest timestamp on a page. For multi-version data, tuple versions valid before the selected timestamp have to be moved and versions still active have to be copied to a new historical node. Historical nodes are immutable and are written in append-only manner to a further storage device. A separator key has to be inserted in the parent node with the lower key on the page and the timestamp of the split. Index records, valid on or after the selected timestamp remain on the page with current data. A Key Split is similar to a split in B$^+$-Trees. The page is split at a split point and a separator key is posted to the parent node. [9]

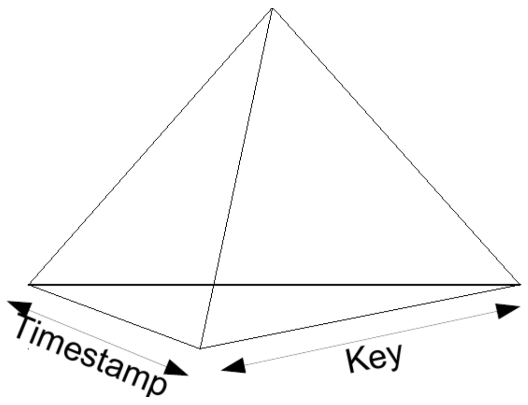

Figure 2: Time-Split B-Tree Index Structure

3.1.2 *Discussion with respect to multi-versioning*

The idea is to separate historical from current data and maintain a history of tuple versions for temporal querying based on transaction time. These kinds of queries are common in temporal DBMS. Inner nodes can be traversed in

Key | Timestamp | Value

Figure 3: Time-Split B-Tree Record Format

both dimensions – key column and timestamp. Traverse steps have to be processed successively in a tree structure. A page is read, the separator key with the pointer to the child page is located and then the child page can be read. This recursive operation is repeated until a leaf page is located and the relevant index record can be fetched. There is high complexity in the traverse steps, which cannot be performed in parallel, so multi-core CPUs are not fully occupied and parallelism of flash is not leveraged optimally. This effect is amplified by range queries in both dimensions. Random reads are performed successively after processing separator keys.

Updates result in a new index record. Older versions are not touched for invalidation, which is principally good for write amplification as well as asymmetry and wear in flash storage media. Also historical versions are written once sequentially on a new historical node. Creating new nodes for separating historical from current data is beneficial in complex memory hierarchies. Data is archived and evicted to a storage medium in memory hierarchy as part of the splitting process. Its final location can be chosen based on the requirements to an ordinary workload, e.g. on fast flash media, if accessing historical data is common, or in other cases on cheap slower devices. However, decision what data is historical or current depends on several factors at split time of overflowing pages, what is done on page level and cannot be adapted to changes in workload. Furthermore, new index records are applied to a random leaf page – a current node. This generates random write I/O all over current nodes at a Time-Split B-Tree, especially, in case of bulk loads and if splits are performed. On persistent storage, the results are in-place updates, high write amplification and random writes, because the whole modified page is written on eviction from main memory.

All versions are maintained in one single tree structure. Since new index records invalidate older ones with same key value, Time-Split B-Trees are qualified for unique keys, but non-unique keys are not supported. Moreover, it is hardly possible to apply changes in key to an index record. If keys are modified, the only possibility is to insert an index record with same key value as its predecessor, which is pointing to an invalid location and a further record with the new key, timestamp and the record id of the successor tuple version. As a result, predecessor and successor versions are unknown in index. Furthermore, index records are sorted by key columns. Since updates in key are performed by inserting two index records with different key, modifications are spread all over current nodes. On eviction from main memory, modified pages are written to persistent storage media. Consequently, random write I/O is performed on flash, what is very costly.

In MV-DBMS it can be mandatory to find predecessor and successor of tuple versions. Time-Split B-Trees do not have this capability. Newer index records invalidate older ones with same key. In best case, the predecessor is located on same page as its successor, but if a Version-Split was performed, traverse steps by timestamp dimension have to be performed on historical data. Especially in case of long living tuple versions, there is a high redundancy, because still valid versions are copied on a Time Split. So multiple random reads occur and storage utilization for archiving is high.

Table 2: Characteristics Review Time-Split B-Tree

Characteristic	Findings
Secondary Storage I/O Pattern	
Write Pattern	Very good append-only behavior for Historical Nodes, but random for Current Nodes with probably high write amplification on eviction from buffer
Read Pattern	Low read amplification, successive random reads on traverse steps
Index Structure and Operations	
Insert Operations	Based on key, random in (un-)cached Current Nodes
Update/Invalidation Operations	Out-of-place insertion of new record with same key, new timestamp and new record id in random (un-)cached Current Nodes, but keys have to be unique
Bulk Loads	Based on distribution of keys, random in (un-)cached Current Nodes
Maintenance Operations	Based on workload, random in (un-) cached Current Nodes and complex split policies
Garbage Collection	In principle, archived Historical Nodes can be removed from index structure with low effort
Redundancy	Probabilistic high redundancy of long living tuple versions
Index Size	One to many records per tuple version with key, timestamp and record id
Multi-Version Capabilities	
Visibility Check	Key and timestamp are sufficient for visibility check
Version Chain Ordering	No connection between successor and predecessor, versions are spread randomly on Historical and Current Nodes
Workload Adaptivity	
Workload Adaptivity	Data placement is based on current workload and split policies – later reorganization is not possible

In summary, Time-Split B-Trees have shortcomings in holding and processing multi-version data as well as leveraging modern hardware. Keys in index have to be unique and it is costly to apply changes in key. Predecessor and successor versions as well as their location are unknown. Probabilistic, a high redundancy of long living versions occur. Random write I/O and high write amplification in current nodes do not fit to characteristics of modern storage media. Pages on flash are written randomly and have to be erased for updates, evoking poor performance and wear.

3.2 Multiversion B-Tree

3.2.1 *Data Structure Overview*

Multiversion B-Trees (MVBT) [1] have the structure of directed acyclic graphs of nodes. Its structure is depicted in Figure 4. Several root nodes are formed while splitting nodes. As a consequence, roots are managed by additional index structures, e.g. a B^+-Tree. Every modification to the indexed dataset generates a new version in a MVBT. A lifespan is maintained at any index record, based on a version for validation and a version for invalidation. Index records in MVBT are formed by key columns, an in-version, an out-version and the record id of a referencing tuple version in base table, as shown in Figure 5. On insertion of a new index record, its in-version is set to current version of the MVBT. Out-version is not set, what means that an index record is still "alive". Deletions set the out-version with the version of the MVBT for invalidation. Updates are performed as a combination of described deletion and insertion operations. An index record belongs to a version, if it is in its lifespan, but less the out-version. In a MVBT, mutable Live Blocks with current data exist as well as immutable Dead Blocks with dataset of a block at the version, when the Dead Block "died". A Live Block becomes a Dead Block, if there is too few space for inserting a new record on an insertion or update operation. In this case, a Version-Split is performed. The old Live Block is copied to a new one, in which invalidated data will be removed. The older one becomes an immutable Dead Block. A new separator key is posted to the parent nodes and the version lifespan of the separator key of the old Dead Block is updated. Eventually, in the new Live Block are too much entries, because most records were still valid on Version-Split. In this case (strong version overflow), a Key-Split is performed. If there are too few valid records (strong version underflow after Version-Split or weak version underflow as a result of invalidations) on a Live Block, a merge with valid records of a sibling Live Block is performed and the sibling becomes a Dead Block. Is there no out-version for invalidation at a record maintained, the record is valid in a Live Block, but was valid at Version-Split of a Dead Block. Live and Dead Blocks are indexed by version lifespan and key range. [1]

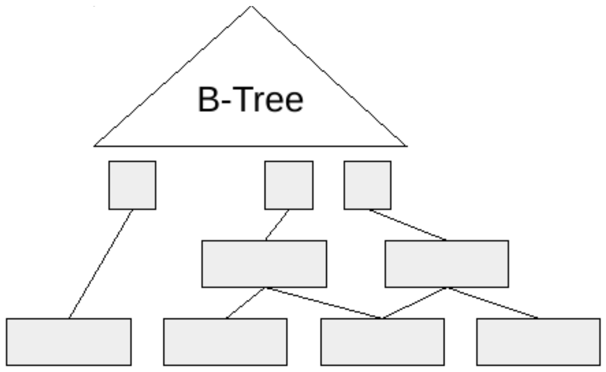

Figure 4: Multiversion B-Tree Index Structure

$$\boxed{\text{Key} \mid V_{in} \mid V_{out} \mid \text{Value}}$$

Figure 5: Multiversion B-Tree Record Format

3.2.2 *Discussion with respect to multi-versioning*

A MVBT is able to manage multi-version data, since a version lifespan is maintained at any index record and separator key. Look-ups benefit from traversing in-version lifespan and key column range, but like in Time-Split B-Trees, the whole traverse step is performed successively with random read I/O and processing separator keys. This operations do not optimally benefit from parallelism in flash and multi-core CPUs.

Updates cause an insertion of a record in a random Live Block as well as an in-place invalidation at the predecessor record. Furthermore, bulk loads affect several Live Blocks, based on distribution of keys. As a consequence, random write I/O is performed all over the Live Blocks in MVBT on eviction from main memory, what causes high write amplification, poor performance and wear on flash. This effect is amplified by complex maintenance operations.

Records provide information about the version of validation and invalidation. In contrast to Time-Split B-Trees, this enables management of non-unique keys in a MVBT. Successor and predecessor can be identified by validation or invalidation version. In best case, predecessors are located on same page like its successor, but based on workload, Version-Splits and modifications in key columns result in fragmentation of versions. So, it is required to traverse a MVBT to find predecessors and successors.

Table 3: Characteristics Review Multiversion B-Tree

Characteristic	Findings
Secondary Storage I/O Pattern	
Write Pattern	Random with high write amplification on eviction from buffer cache, finally for Dead Blocks
Read Pattern	Low read amplification, successive random reads on traverse steps
Index Structure and Operations	
Insert Operations	Based on key, random in (un-)cached Live Blocks
Update/Invalidation Operations	In-place invalidation with timestamp and out-of-place insertion of new record with new key, timestamp and record id in random (un-)cached Live Blocks
Bulk Loads	Based on distribution of keys, random in (un-)cached Live Blocks
Maintenance Operations	Based on workload, random in (un-) cached Live Blocks and complex split policies
Garbage Collection	Dead Blocks are spread randomly, high effort
Redundancy	Probabilistic high redundancy of long living tuple versions
Index Size	One to many records per tuple version with key, in-version, out-version and record id
Multi-Version Capabilities	
Visibility Check	Key, in-version and out-version are sufficient for visibility check
Version Chain Ordering	Connection of successor and predecessor by MVBT version, versions are spread randomly on Live and Dead Blocks
Workload Adaptivity	
Workload Adaptivity	Data placement is based on current workload and split policy – later reorganization is not possible

Dead Blocks are written finally on eviction with all valid and invalidated versions of a Version-Split. In case of long living tuple versions, a high redundancy arises. This causes one random write I/O and additional storage costs on flash. In fact of creating a new Live Block on Version-Split, Dead Blocks are spread randomly all over persistent storage media. High complexity on archive or garbage collection operations arises. Furthermore, whole data placement is based on split policy and no possibility for reorganization along complex memory hierarchies is provided.

To sum up, MVBT provide fast look-up of multi-version data, but is not optimal for modern hardware, especially in case of high update rates. MVBT does not leverage complex memory hierarchies. Modifications are random in Live Blocks and result in complex maintenance operations. Likely, every modification cause in-place updates in Live Blocks, which are applied to persistent storage media on eviction. Random write I/O, high write amplification and wear occur on flash storage media.

3.3 MV-IDX

3.3.1 Data Structure Overview

A MV-IDX [3] is a lightweight multi-version indexing approach. It is based on a regular B$^+$-Tree, but maintains a virtual identifier (VID) at any index record instead of the tuple id of its referenced tuple version in base table (see Figure 7). A VID is maintained for every tuple. Tuple versions of a tuple do have the same VID. As depicted in Figure 6, VIDs are maintained in a VID-List located in main memory. For any VID in the VID-List, one Data Node is maintained for every tuple version. The VID-List points to the newest Data Node of a tuple. A Data Node provides a timestamp of its creating transaction, the tuple id in base table of a tuple version, a flag, if the version is committed and a pointer to the Data Node of its predecessor. Only Data Nodes of possibly visible versions is maintained in a VID-List and the B$^+$-Tree structure. In this data structures, all required information is provided to perform an index-only visibility check. [3]

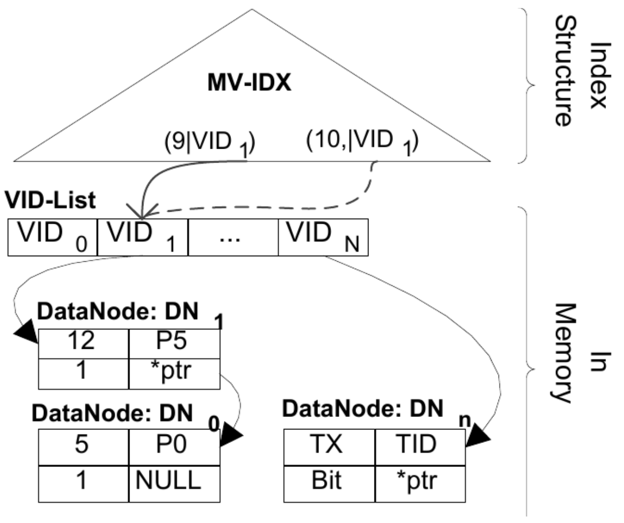

Figure 6: MV-IDX Index Structure

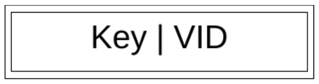

Figure 7: MV-IDX Record Format

3.3.2 Discussion with respect to multi-versioning

The index structure in a MV-IDX bases on a B$^+$-Tree. Index records are inserted in a random leaf page, sorted by its key columns. This behavior is amplified in case of bulk loads. As a result, random maintenance operations occur all over the index structure. On eviction of modified pages from main memory, random write I/O with high write amplification occur on persistent storage media. In relation to a B$^+$-Tree, amount of inserted versions can be reduced. It is necessary to insert a new index record in its B$^+$-Tree component, if the key of a new version changes. If indexed columns are stable and a record is maintained, it is sufficient to insert a new Data Node in the VID-List on update, because the key and VID of a tuple are already maintained in the index structure. Deletes can be handled similar by invalidation in Data Nodes.

Look-ups and Scans are performed on key columns. Therefore, traverse steps are performed successively on random nodes. In a MV-IDX index structure, any possibly visible version is indexed by key without any information for visibility checks. Information to perform visibility checks are provided at the Data Nodes in the VID-List in main memory. The problem is, timestamps of versions are not considered in traverse steps. Any index record, matching to a search key, has to be fetched and its Data Nodes in VID-List have to be consulted. As a result, a huge amount of invisible data to a transaction is read. The more versions are held, the more records are read and checked for visibility in its in-memory components. A MV-IDX can handle multi-version data with less updates on keys very well, because of its in-memory version chain, but is not suitable for archiving or temporal querying.

Furthermore, the VID-List and its Data Nodes are expected to be held in main memory. This behavior has two problems. First, a mechanism is required to get in-memory components crash safe. Logging of B^+-Tree operations is not sufficient. Second, modifications while ongoing runtime let its in-memory data structures grow. It is necessary to perform garbage collection of old versions in main memory components and B^+-Tree structure to reclaim space. This causes updates and maintenance operations in whole B^+-Tree structure and results in random write I/O on persistent storage media. One solution could be to persist its in-memory components, what results in additional complexity and random I/O.

Table 4: Characteristics Review MV-IDX

Characteristic	Findings
Secondary Storage I/O Pattern	
Write Pattern	Random with high write amplification on eviction from buffer cache
Read Pattern	Probably invisible records are read, successive random reads on traverse steps
Index Structure and Operations	
Insert Operations	Based on key, random in (un-)cached Nodes, new Data Node in VID-List
Update/Invalidation Operations	In-memory append of Data Node, if the attribute value did not change, else insertion of a new record with new key and same VID in random (un-)cached Leaf Node and append of Data Node
Bulk Loads	Based on distribution of keys, random in (un-)cached Nodes and new Data Nodes in VID-List
Maintenance Operations	Based on workload, random in whole B^+-Tree structure
Garbage Collection	Leaf Nodes are spread randomly, but GC of Data Nodes in VID-List is cheap
Redundancy	Very low redundancy, at most one record per tuple version
Index Size	One record per tuple with same key and VID and one Data Node for every tuple version in VID-List
Multi-Version Capabilities	
Visibility Check	Key and Data Nodes are sufficient for visibility check
Version Chain Ordering	Optimal connection of successor and predecessor in Data Nodes of VID-List
Workload Adaptivity	
Workload Adaptivity	Data placement is based on current workload – later reorganization is not possible

In a short resume, a MV-IDX brings benefits in low complexity of traverse steps, relative good query and scan performance, if most scanned data is visible to a transaction, and versions are well connected in Data Nodes of its VID-List. Updating versions without changing key columns can be performed in-memory on Data Nodes in its VID-List and no additional maintenance is required in its B^+-Tree structure. Information stored in MV-IDX is sufficient to perform a visibility check. There are shortcomings in temporal querying and additional in-memory structures have to become crash safe. Updates on key columns in a new version result in random update and maintenance operations all over the B^+-Tree structure and random write I/O as well as high write amplification and wear on persistent storage media, like flash. Moreover, there is no reasonable possibility to spread data along complex memory hierarchies. It is fair to say, MV-IDX leverages the characteristics modern hardware to a partial degree.

3.4 Bi-Temporal Timeline Index

3.4.1 Data Structure Overview

The Bi-Temporal Timeline Index [6], depicted in Figure 8, is based on the Timeline Index [7] and extends the former one by supporting additional application time dimensions, which are not maintained by the DBMS. A Timeline Index preserves an overview of visible tuple versions at any point in transaction time dimension. Every tuple version creation and invalidation is listed in an Event List. A Version Map is able to reconstruct visible tuple versions at any point in transaction time. A new version can be created in the Version Map and mapped to a span of entries in Event List, e.g. on transaction commit. This data structures and maintained information (see Figure 9) are sufficient to perform multi-version indexing and temporal querying. Effort of reconstructing visible tuple versions underlay a linear growth. A (Bi-Temporal) Timeline Index performs checkpoints to counteract linear growth of reconstruction costs. A checkpoint is, apart from the regular term of checkpoints in DBMS, an independent operation and can be performed in different frequencies. When a checkpoint is performed, a Visibility Map is created, which contains the visibility information of any indexed tuple in a bit-vector. In a temporal query, the latest created Visibility Map before the requested time slice in transaction time can be checked for visibility of a tuple at this point in transaction time. Moreover, all events in the Event List created after this checkpoint have to be checked until the visibility of the desired version in Version Map, matching to the requested time slice, is reconstructed. Linear growing effort of visibility reconstruction is reduced to events created after the latest checkpoint in transaction time. A Bi-Temporal Timeline Index maintains an Event Map and Visibility Bitmaps for any application time dimension at any checkpoint. [6]

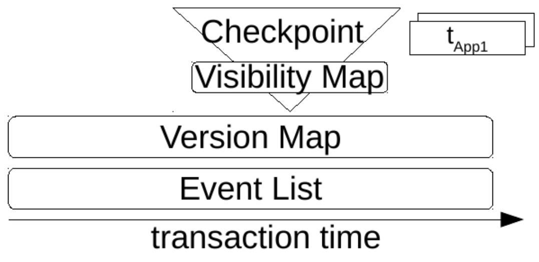

Figure 8: Bi-Temporal Timeline Index Structure

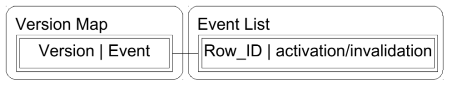

Figure 9: Bi-Temp. Timeline Index Record Format

3.4.2 Discussion with respect to multi-versioning

Bi-Temporal Timeline Indexes feature transaction and application time dimensions. Mainly, it is designed for column-oriented in-memory DBMS, even its concepts can be applied to row-oriented DBMS [6]. Therefore, all required data structures have to be integrated in the DBMS and become crash save. In [6] the data structures are implemented in additional tables in SAP-HANA. Visibility Maps, Version Map and Event List have to be processed sequentially to reconstruct visibility of a tuple at any point in transaction and application time dimension. Key columns, like in the ubiquitous B$^+$-Tree are not maintained in the index. Required columns are looked up in base tables, performed by algorithms of the in-memory DBMS. Column-oriented in-memory DBMS benefit from this behavior twofold. First, required size of the index is low and more data can be located in main memory. Second, matured in-memory algorithms are used for look-up of required attributes. In row-oriented DBMS, there are some drawbacks. Whole tuples have to be fetched from base table, albeit not all attributes are required to answer a query. Traditional indexes maintain key columns of tuple attributes in frequently performed queries. As a result, unnecessary data is read from base tables randomly. Furthermore, data is not sorted by key columns. This problems can be solved by maintaining an additional index, e.g. an ubiquitous B$^+$-Tree. Benefits in storage size are wasted. Furthermore, additional maintenance effort occurs in the additive index structure. Random maintenance operations in B$^+$-Trees result in poor performance on persistent storage media, like flash.

Transaction time is strictly advancing over time. Inserts, updates and bulk loads in a Bi-Temporal Timeline Index data structures are performed append-only. Events in Event List as well as versions in Version Map are appended and won't be modified. Visibility Maps for transaction time as well as Event Maps and Visibility Bitmaps for any further application time dimension are calculated once on any checkpoint. This behavior enables sequential writes on persistent storage media. Flash benefits in several ways. First, sequential writes are performed very fast. Second, write amplification is low, data is written once and won't be modified. This behavior reduces wear on flash.

Furthermore, based on workload and data placement in complex memory hierarchies, reads on slower storage media are rare for multi-version indexing. The Bi-Temporal Timeline Index requires all events in Event List since latest checkpoint in transaction time before the queried version in Version Map and the Visibility Bitmap created on checkpoint to restore visibility of a tuple. Events created before this checkpoint or after the requested version are not required. If some data structures are not located in main memory, it is possible to fetch them in parallel from flash.

Some redundancy occurs on checkpoints. Visibility Maps and Event Lists for application time dimensions are calculated and stored for every checkpoint. Size of bit maps is relatively low, but gets a factor on frequently created checkpoints. The less checkpoints were performed, the more sequential processing effort is required to calculate visibility of a tuple.

In a nutshell, Bi-Temporal Timeline Indexes have near optimal characteristics for modern hardware. Write I/O on persistent storage media can be performed sequentially, if required. Temporal information is stored efficiently in several data structures. Visibility of any tuple can be restored in every transaction and application time dimension with linear growing computing effort. However, key columns of frequently queried tuple attributes are not maintained in the proposed data structures. Sorting and look-up operations are delegated to further DBMS layers. In column-oriented in-memory DBMS this behavior is beneficial, but not optimal in row-oriented DBMS. Either whole tuples have to be read and sorted from base tables, what result in enormous querying effort and excessive read on persistent storage devices for table scans, or an additional index structure must be maintained. In this case, additional maintenance effort and storage utilization is required.

Table 5: Characteristics Review Bi-Temporal Timeline Index

Characteristic	Findings
Secondary Storage I/O Pattern	
Write Pattern	Possible sequential append-only of events
Read Pattern	Possible sequential reads of events from latest checkpoint
Index Structure and Operations	
Insert Operations	Append-only of Events and Version
Update/Invalidation Operations	Append-only of Events and Version
Bulk Loads	Append-only of Events and Version
Maintenance Operations	Checkpoints with Visibility Map for transaction time and Event Maps and Visibility Bitmaps for any application time dimension
Garbage Collection	Possibility to remove Events before Checkpoints – very cheap
Redundancy	Visibility Maps at any Checkpoint
Index Size	Very small, Version Map, Events in Event-List and Checkpoints
Multi-Version Capabilities	
Visibility Check	Only Events are stored, attributes have to be fetched from base tables
Version Chain Ordering	Every version can be restored from Checkpoints and Event-List processing
Workload Adaptivity	
Workload Adaptivity	Suitable to high update rates, but reorganization is not possible

4. EXTENDED SCOPE

4.1 LSM-Trees

LSM-Trees [12] and bLSM-Trees [13] are popular log-based indexing approaches for high update rates in K/V-Stores, which leverage characteristics of asymmetric storage media, like SSDs. Unfortunately, similar to the ubiquitous B$^+$-Tree, there is no support for multi-version data. Because of their high relevance in the database community, we give a short overview of mentioned log-based indexing approaches and a discussion with respect to their suitability to modern hardware.

4.1.1 Data Structure Overview

LSM-Trees are composed of several components of different size. Each component has its own tree-based sorted data structure for indexing. A smaller one is always located in main memory, which is called the *C0* component. Further components (*C1* to *Cn*) resident on secondary storage media in ascending capacity. Data modifications, like inserts and deletes, are performed in the cached *C0* component without any latencies for reading a page from secondary storage. Inserted data in a random order is transformed in a eviction of pages with data in presorted order. If the size of a component exceeds a certain threshold, a merge with the next larger component has to be performed. Data of evicted pages is merged with data in the next larger component and is written to a new sequence of blocks with a fill factor of 100 percent. This enables a LSM-Tree to write data sequentially in a log-structured manner. A look-up affects every component, especially in case of range queries, beginning with the *C0* component. However, in case of a point query, the algorithm can break, if a matching record was found. [12]

An enhancement of LSM-Trees for steadiness in performance and write optimization is the bLSM-Tree. Bloom filters protect components against point queries, however, they are ineffective in case of scans. Each component has to be looked-up, so there is a fixed number of three components. No further component is created in case of load spikes to the *C0* component. Write optimization is achieved by increasing the effective amount of data per merge with replacement selection. Furthermore, there is a matching in the scheduler between insertion rates to the *C0* component, merges between *C0* and *C1* components, as well as between *C1* and *C2* components, whereby steals are avoided and throughput is optimized. [13]

4.1.2 Discussion with respect to suitability to modern Hardware

LSM- and bLSM-Trees are optimized for high update rates as well as bulk loads and achieve sequential write patterns to secondary storage media by presorting inserted and updated data in the *C0* component, which is located in main memory, and merge operations between all components. No B^+-Tree-like update in-place operations are performed on already evicted pages. Evicted pages have a fill factor of 100 percent, but some redundancy occurs on updates. This behavior is very beneficial for traditional as well as modern storage media. However, for asymmetric storage media, write amplification of merge operations is not optimal. Merge operations between a smaller component C_s and the next larger component C_l need to read all affected data of both components in a predefined key range, what can be handled in asymmetric storage media very well. A subset of keys of C_s and C_l are processed in a merge sort operation and the result will be appended in the C_l component sequentially. If the merge operation succeeds, the blocks of the subset can be freed. With evolving time, merge operations will affect subsets of keys several times. The sequential write pattern will lower the write amplification (in comparison to a B^+-Tree), but it is increased by overlapping merge operations.

While merge operations, garbage collection can be performed. Updating a value of a record, which is located in a component on secondary storage, result in a new version of the record in the *C0* component. Merging key ranges in components enable garbage collection of older versions. Nevertheless, aged versions stay long time in larger components and have to be processed on range scans. In case of multi-version data, several problems occur. It is not save to garbage collect versions without checking visibility in base tables, because they are maybe still visible to active transactions, especially in case of smaller components, which contain newer versions. Furthermore, scan performance will slow down, when accessing larger components, where lots of versions tent to be invisible to active transactions, but have to be rechecked for visibility in base tables, too.

Read amplification is increased by the number of components. In theory, asymmetric storage media can handle increased reading amounts very well, due to low read latencies and high parallelism. Traversing a tree structure cannot fully benefit from parallelism, because next required pages are determined after processing a page. LSM- and bLSM-Trees maintain a tree-based index structure for every component. The hight of every tree-based component is logarithmic to its capacity. In comparison to a B^+-Tree, more inner nodes are required and have to be processed in traversal steps. LSM-Trees can break look-up algorithm, if a matching key was found. Newer data is located in smaller and first processed components. Furthermore, bloom filters protect *C1* and *C2* components in bLSM-Trees against unnecessary querying for point queries. However, range queries require to look-up every component.

LSM and bLSM-Trees are able to share components along several storage media in memory hierarchy. Nevertheless, data placement is based on current insert and update workload and on capabilities of merge operations. Recently used key ranges are maybe not evicted and merged with further components, so they can stay in main memory. However, data placement cannot be optimized for current look-ups. It is fair to say that LSM- and bLSM-Trees not fully leverage complex memory hierarchies.

To sum up, LSM- and bLSM-Trees are a good choice for high update rates, even if their write amplification is not optimal. Their read behavior brings some drawbacks due to separate index structures for every component and data organization is not optimal. Even if there are several versions of a record in different components, because updates are performed out-of-place in *C0* component and maybe merged in merge steps, they are not able to perform a visibility check and are not optimal for indexing multi-version data.

4.2 Bw-Trees

Bw-Trees [11] are a highly scalable indexing approach that is popular in main memory DBMS. They leverage characteristics of modern processing and storage technologies by latch-free log-based index operations. Unfortunately, Bw-Trees have no multi-version capabilities. There are possibilities to combine multi-version indexing approaches, like Time-Split B-Trees, with latch-free operations of a Bw-Tree, for example in the TSBw-Tree [10]. This will enable faster index operations, but does not solve all problems outlined in Section 3.1. We give an overview of the data structure and a short discussion to the Bw-Tree on modern hardware technologies.

4.2.1 Data Structure Overview

Bw-Trees consist of inner nodes as well as leaf nodes, which contain records, similar to a B^+-Tree and an in-memory mapping table for logical page pointers. Data modifications and index operations are appended with delta records in a singly linked list. No latches are required, because modifications are atomic compare and swap (CAS) operations. As a result, less cache invalidations occur in a multi-core CPU. Entries in the mapping table point to pages or to delta records. Delta records can point to further delta records or the page. When requesting a page, the whole linked list, from the mapping table via all appended delta records through the page is processed and the current logical state of the "elastic" page can be determined. Leaf nodes are linked via logical page pointers, whereby siblings are requested with help of the mapping table. Splits and merges are performed in multiple atomic steps with CAS operations. In case of a split, a new sibling node is created in main memory and an entry is added in the mapping table. Afterwards, a split delta record is appended to the old page and the pointer in the mapping table is set to the delta record in a CAS operation. The split delta record points to the old page and there is a sibling pointer to the new page. Finally, an index entry delta is added to their parent node that points to the parent node and the new page. Merges are performed similar with help of delta records and CAS operations. Eviction of pages to secondary storage is in a log-based append-only manner, no in-place updates are required. Delta records are batched and written out-of-place. Fetching an "elastic" page requires the mapping table and to read all delta records and the page itself from secondary storage. Epochs and consolidating delta records enable garbage collection. [11]

4.2.2 *Discussion with respect to suitability to modern Hardware*

Bw-Trees are optimized for modern hardware technologies and enable high scalability and concurrency, due to latch-free log-based data modification and index operations. This behavior enables a high CPU cache hit rate, because very less cache invalidations occur. Processing units are kept busy and do not idle.

Their write pattern to secondary storage is in a sequential append-only manner. Write amplification is very low, because pages and delta records are written once and will not change, except for garbage collection. Reads from secondary storage are random, because delta records have to be read, but asymmetric storage media can handle reads very well, due to high parallelism and low read latencies, e.g. on flash. Based on length of the delta chain, parallelism is not fully leveraged, because all used pages on secondary storage have to be read successively in a linked list. This problem is reduced by batching delta records. After reading a page, a fraction of the delta chain can be processed, before reading a further page. Furthermore, garbage collection reduces the length of delta chains. However, if it is common that "elastic" pages get evicted after a few updates many times, the delta chain will be fragmented or high effort on garbage collection arise.

Leveraging several levels in complex memory hierarchy seems to be a problem for Bw-Trees. It is possible to optimize write behavior of delta records to secondary storage, e.g. if delta records can be batched on non-volatile memories, but it is hard to reorganize data for current read workload. In this latch-free indexing approach, data placement mainly depends on current data modifications. Bw-Trees are optimized for huge amounts of main memory, whereby writes and reads on secondary storage are rare and it is common that most required data is located in CPU caches and main memory.

Managing multi-version data is not supported in Bw-Trees. Like in B^+-Trees, candidates returned by the index scan have to be validated in base tables. In theory every delta record corresponds to a version of a tuple, so timestamps could solve this problem. Newer versions can be found at the beginning of a delta record chain of a page, but to determine current state of the "elastic" page, the whole chain has to be processed first. There would be no separation of current and historical data – read and scan performance would decrease. One idea is the TSBw-Tree [10], which combines the latch-free log-based index operations of the Bw-Tree with the good temporal but limited multi-version capabilities of a Time-Split B-Tree (outlined in Section 3.1).

In summary, the Bw-Tree provides high scalability, parallelism and concurrency and leverages modern hardware technologies very well for main memory DBMS. Its write pattern on secondary storage is near optimal, but reads of delta record chains do not fully leverage parallelism in asymmetric storage media. Reorganization methods for current read workloads in complex memory hierarchies are not provided. Bw-Trees do not have multi-version capabilities, but it is possible to combine their in-memory approaches with tree-based indexes, which can handle multi-version data.

5. FUTURE RESEARCH

The challenge is to combine multi-version capabilities in one single index structure, which is able to leverage characteristics of modern hardware and is aware of complex memory hierarchies as well as provides a workload adaptivity. Partitioned B-Trees [4] could be an approach that is able to fulfill mentioned requirements. A Partitioned B-Tree is based on a B^+-Tree. A partition number is added to any index record in an artificial leading key column. This enables partitioning of data in one single index structure that preserves an alpha numeric sort order. The structure of a Partitioned B-Tree is depicted in Figure 10. Index records of tuples can be appended in a new partition and written out sequentially to secondary storage on eviction of a whole partition from main memory. Bulk loads can be performed in an additional partition, without affecting further partitions, which handle current workloads in parallel. On load spikes of current workload, it is possible to throttle or stop the bulk load operation without wasting work and continue at a later point in time [4]. Evicted partitions become immutable, except for garbage collection or reorganization operations, so write amplification is very low and there is no need for random write I/O. Its write pattern can be designed optimal for modern storage technologies.

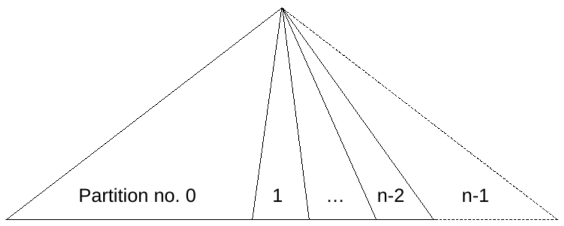

Figure 10: Partitioned B-Tree Index Structure

Record Type | *Timestamp* | Partition Number | Key | Value

Figure 11: Partitioned B-Tree Record Format

Furthermore, Partitioned B-Trees are able to serve as version store in MVCC, by maintaining a timestamp at index records either for validation or invalidation [4]. One timestamp is sufficient, if a deletion marker ("anti matter") is inserted out-of-place in a successor partition. The "anti-matter" can be implemented as a single record type and/or as a replacement record, that also set the validity of the new tuple version, similar to Time-Split B-Trees [9]. An index record consists of meta data, e.g. the record type and the timestamp, partition number and key value as columns and the record id as value (see Figure 11). Read characteristics in flash are leveraged by processing several partitions in parallel. As a result, there is less idle in processing units. Furthermore, recently inserted index records stay in main memory, whereby cores in CPU are kept busy, if look-ups are delegated to newest partitions first. Therefore, a latch-free environment in partitions in main memory, like Bw-

Trees [11], is beneficial. Immutable persisted partitions enable the use of FPGAs, which allow fast and energy efficient processing of huge amounts of data near its storage location. Additional, persistent partitions can be protected from unnecessary processing by filter technologies. Workload adaptivity is provided by Adaptive Merging [5], by what index records can be placed in partitions located along storage devices in complex memory hierarchies, as desired. Partitioned B-Trees seem to be a multi-version indexing approach leveraging characteristics of modern hardware technologies.

Table 6: Characteristics Assumptions Partitioned B-Trees

Characteristic	Assumption
Secondary Storage I/O Pattern	
Write Pattern	Sequential append-only of index records in a partition
Read Pattern	Successive for inner nodes, which are well cached for partitions with newly inserted records, and high parallelism in leaf nodes
Index Structure and Operations	
Insert Operations	Inserts in leaf nodes of partition in main memory
Update/Invalidation Operations	Out-of-place in leaf nodes of further partitions in main memory
Bulk Loads	Append in additional partition in main memory
Maintenance Operations	Maintenance is reduced to index nodes in main memory – inner nodes, which directing to leaf nodes of not yet evicted partitions and its leaf nodes
Garbage Collection	Records of tuple versions, that are not visible to any running transaction, can be removed in main memory, if the partition was not yet evicted, else the records can be garbage collected as part of merges of partitions
Redundancy	Low redundancy with exact one index record per indexed tuple version and up to one "anti-matter" for invalidations
Index Size	One or two index record with Partition number, key columns, one timestamp (validation and/or invalidation) and record id – possibility of compression
Multi-Version Capabilities	
Visibility Check	Key and timestamps in index records and "anti-matter" are sufficient for visibility check as part of the search algorithm
Version Chain Ordering	Connection of successor and predecessor versions located in consecutive partitions by key and timestamp
Workload Adaptivity	
Workload Adaptivity	Suitable to high update rates as well as reorganization by Adaptive Merging

6. CONCLUSION

This paper outlines popular indexing approaches for multi-version and temporal data. The objective is to review their capabilities for processing and managing multi-version data in modern computing and storage technologies and complex memory hierarchies. We established that all of the outlined approaches have drawbacks in at least one investigated area. Multi-version and temporal index structures presented in Section 3 have shortcomings in multi-version capabilities, complexity in look-up and maintenance, alignment to modern computing and storage technologies or in their specialization for single fields of application. Indexing approaches presented in Section 4, which can leverage characteristics of modern hardware technologies, do not have multi-version capabilities. The challenge is to combine multi-version capabilities in one single index structure, which is able to handle characteristics of modern hardware and is aware of complex memory hierarchies as well as provide a workload adaptivity.

Partitioned B-Trees seems to be a good indexing approach for multi-version data and modern hardware technologies.

7. ACKNOWLEDGMENTS

The authors wish to thank Goetz Graefe for the comments on this research. This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program 'Services Computing'.

8. REFERENCES

[1] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *The VLDB Journal*, 5(4):264–275, Dec. 1996.

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

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

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

[5] G. Graefe and H. Kuno. Adaptive indexing for relational keys. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 69–74, March 2010.

[6] M. Kaufmann, P. M. Fischer, N. May, C. Ge, A. K. Goel, and D. Kossmann. Bi-temporal timeline index: A data structure for processing queries on bi-temporal data. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 471–482, 2015.

[7] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May. Timeline index: A unified data structure for processing queries on temporal data in sap hana. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1173–1184, New York, NY, USA, 2013. ACM.

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

[9] D. Lomet and B. Salzberg. The performance of a multiversion access method. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 353–363, New York, NY, USA, 1990. ACM.

[10] D. B. Lomet and F. Nawab. High performance temporal indexing on modern hardware. In J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, editors, *ICDE*, pages 1203–1214. IEEE Computer Society, 2015.

[11] D. B. Lomet, S. Sengupta, and J. J. Levandoski. The bw-tree: A b-tree for new hardware platforms. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 302–313, Washington, DC, USA, 2013. IEEE Computer Society.

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

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