The VLDB Journal (2016) 25:651–672
DOI 10.1007/s00778-015-0410-5

SPECIAL ISSUE PAPER

Exploiting SSDs in operational multiversion databases

Mohammad Sadoghi[1] · **Kenneth A. Ross**[1,2] · **Mustafa Canim**[1] · **Bishwaranjan Bhattacharjee**[1]

Received: 29 January 2015 / Revised: 18 September 2015 / Accepted: 26 October 2015 / Published online: 17 November 2015

Abstract Multiversion databases store both current and historical data. Rows are typically annotated with timestamps representing the period when the row is/was valid. We develop novel techniques to reduce index maintenance in multiversion databases, so that indexes can be used effectively for analytical queries over current data without being a heavy burden on transaction throughput. To achieve this end, we re-design persistent index data structures in the storage hierarchy to employ an extra level of indirection. The indirection level is stored on solid-state disks that can support very fast random I/Os, so that traversing the extra level of indirection incurs a relatively small overhead. The extra level of indirection dramatically reduces the number of magnetic disk I/Os that are needed for index updates and localizes maintenance to indexes on updated attributes. Additionally, we batch insertions within the indirection layer in order to reduce physical disk I/Os for indexing new records. In this work, we further exploit SSDs by introducing novel DeltaBlock techniques for storing the recent changes to data on SSDs. Using our DeltaBlock, we propose an efficient method to periodically flush the recently changed data from SSDs to HDDs such that, on the one hand, we keep track of every change (or delta) for every record, and, on the other hand, we avoid redundantly storing the unchanged portion of updated records. By reducing the index maintenance overhead on transactions, we enable operational data stores to create more indexes to support queries. We have developed a prototype of our indirection proposal by extending the widely used generalized search tree open-source project, which is also employed in PostgreSQL. Our working implementation demonstrates that we can significantly reduce index maintenance and/or query processing cost by a factor of 3. For the insertion of new records, our novel batching technique can save up to 90 % of the insertion time. For updates, our prototype demonstrates that we can significantly reduce the database size by up to 80 % even with a modest space allocated for DeltaBlocks on SSDs.

Keywords Multiversion databases · SSD · Flash storage · Index maintenance

✉ Mustafa Canim
mustafa@us.ibm.com

Mohammad Sadoghi
msadoghi@us.ibm.com

Kenneth A. Ross
kar@cs.columbia.edu

Bishwaranjan Bhattacharjee
bhatta@us.ibm.com

1 IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

2 Columbia University, New York, NY, USA

1 Introduction

In a multiversion database system, new records do not physically replace old ones. Instead, a new version of the record is created, which becomes visible to other transactions at commit time. Conceptually, there may be many rows for a record, each corresponding to the state of the database at some point in the past. Very old versions may be garbage-collected as the need for old data diminishes, in order to reclaim space for new data.

When indexing data, one typically indexes only the most recent version of the data, since that version is most commonly accessed. In such a setting, record insertions, deletions, and updates trigger I/O to keep the indexes up to date. With a traditional index structure, the deletion of a record

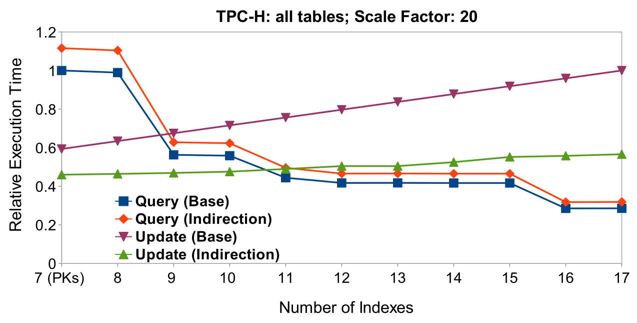


Fig. 1 Effect of adding indexes in operational data stores

requires the traversal of each index and the removal of the row-identifier (RID) from the leaf node. The update of a record (changing one attribute value to another) creates a new version, triggering a traversal of all indexes to change the RIDs to the new version's RID.[1] For a newly inserted record, the new RID must be inserted into each index. Indexes may be large, and in aggregate much too large to fit in the RAM bufferpool. As a result, all of these index maintenance operations will incur the overhead of physical I/O on the storage device.

These overheads have historically been problematic for OLTP workloads that are update-intensive. As a result, OLTP workloads are often tuned to minimize the number of indexes available. This choice makes it more difficult to efficiently process queries and to locate records based on secondary attributes. These capabilities are often important for operational data stores [35]. For example, it is not uncommon to find tens of indexes to improve analytical and decision-making queries even in TPC benchmarks [19,20] or enterprise resource planning (ERP) scenarios [13,14].

Our goal is to reduce the overhead of index updates, so that indexes can be used effectively for analytical query processing without being a heavy burden on transaction throughput. The query versus update dilemma is clearly captured in Fig. 1, a preview of our experimental results. The execution time for analytical queries is reduced as more indexes are added. However, this reduction comes at the cost of increasing the update time in the Base approach. In contrast, by employing our technique (the Indirection approach) the incurred update cost is significantly smaller.

To address this dilemma, we utilize a solid-state storage layer. Based on current technologies, solid-state disks (SSDs) are orders of magnitude faster than magnetic disks (HDDs) for small random I/Os. However, per gigabyte, SSDs are more expensive than magnetic disks. It therefore pays to store the bulk of the data on magnetic disks, and to reserve the SSD storage for portions of the data that can benefit the most, typically items that are accessed frequently and randomly. We describe a prototype of our Indirection approach based on the widely used generalized search tree (GiST) package [29,31], which is also employed in industrial-strength open-source database management systems such as PostgreSQL [6], PostGIS [5], OpenFTS [4], and BioPostgres [1].

Unlike previous approaches [15,17,18,23,37], we do not propose to simply store "hot" data on SSDs. Instead, we change the data structures in the storage hierarchy to employ an extra level of indirection (an earlier version of this paper first introduced in [53]) through solid-state storage. Because the solid-state storage is fast, the extra time incurred during index traversal is small, as we demonstrate experimentally. The extra level of indirection dramatically reduces the amount of (magnetic disk) I/Os that are needed for index updates; thus, making our Indirection technique algorithmically superior to the existing techniques irrespective of the underlying storage technologies employed. Furthermore, our Indirection technique can rely exclusively on fast SSD I/Os to support deletions and updates (even when both data and indexes are disk-resident), with the exception of indexes on changed attributes. We can also reduce the magnetic disk I/O overhead for insertions using our proposed LIDBlock. In this paper, we extend [53] by introducing our novel DeltaBlock techniques. These techniques provide an efficient means to access and reconstruct the latest version of records by consulting recent updates (i.e., deltas batched on SSDs) while naturally reducing and compressing data by periodically flushing deltas from SSDs to HDDs. While we describe our techniques in terms of SSDs, we are not limited to a disk-like form factor. In fact, alternative form factors (e.g., FusionIO auto-commit memory [27]) with smaller I/O granularities would provide even better performance because our proposed updates on solid-state storage are small.

Another potential advantage of our work—efficient index maintenance of multiversion databases—is to support multiversion concurrency control (MVCC). The MVCC model is being revived [6,32,38,39,52] mostly due to the increased concurrency available in modern hardware with large main memories and multicore processors. However, this increased concurrency comes at the cost of increased locking contention among concurrent read/update queries, which could be alleviated using optimistic concurrency control over a multiversion database [38,52].

1.1 Multiversion databases

By retaining old data versions, a system can enable queries about the state of the database at points in the past. The ability to query the past has a number of important applications [56], for example: (1) a financial firm is required to retain any changes made to client information for up to 5 years in accordance with auditing regulations; (2) a retailer ensures that only one discount for each product is offered

[1] In the case of the modified attribute, the position of the record in the index may also change.

at any given time; (3) a bank needs to retroactively correct an error for miscalculating the promised introductory interest rate. In addition to these business-specific scenarios, there is an inherent algorithmic benefit for retaining the old versions of the record and avoiding in-place changes, that is, to enable efficient optimistic locking and latch-free data structures.

A simple implementation of a multiversion database would store the row-identifier (RID) of the old version within the row of the new version, defining a linked list of versions. Such an implementation allows for the easy identification of old versions of each row, but puts the burden of reconstructing consistent states at particular times on the application, which would need to keep timing information within each row.

To relieve applications of such burdens, a multiversion database system can maintain explicit timing information for each row. In a valid time temporal model [28] each row is associated with an interval [begin-time, end-time) for which it was/is current. Several implementation choices exist for such a model. One could store the begin-time with each new row and infer the end-time as the begin-time of the next version. Compared with storing both the begin-time and end-time explicitly for each row, this choice saves space and also saves some write I/O to update the old version. On the other hand, queries over historical versions are more complex because they need to consult more rows to reconstruct validity intervals.

A *bitemporal* database maintains two kinds of temporal information, the system (i.e., transaction) time, as well as the application time (sometimes called "business time").

In this work, we do not commit to any one of these implementation options, each of which might be a valid choice for some workloads. For any of these choices, our proposed methods will reduce the I/O burden of index updates. Some of our techniques, such as the LIDBlock and DeltaBlock, apply to both versioned and non-versioned databases.

1.2 Physical organization

There are several options for the physical organization of a temporal database. A complete discussion of the alternatives is beyond the scope of this paper. We highlight two options that have been used in commercial systems, namely, the *history-table* versus the *single-table* approach.

One organization option appends old versions of records to a *history table* and only keeps the most recent version in the main table, updating it in-place. Commercial systems have implemented this technique: In IBM DB2 it is called "System-period data versioning" [34], and it is used whenever a table employs transaction time as the temporal attribute. The Oracle Flashback Archive [49] also uses a history table. Such an organization clusters the history table by end-time and does not impose a clustering order on the main table. Updates need to read and write the main table, and also write to the end of the history table. Because updates to the main table are in-place,[2] an index needs to be updated only when the corresponding attribute value changes. For insertions and deletions, all indexes need to be updated. In short, using the history table approach (1) the temporal ordering of the data is lost; (2) additional random I/Os are required to perform in-place updates of records; (3) the number of database objects (e.g., tables, indexes, and constraints) are potentially doubled, which increases the overall management and maintenance cost of the database and slows down the query optimization runtime; and (4) a less effective query plans are constructed for certain temporal range queries that are forced to union the history and the main tables instead of using range-partitioned tables for maintaining the single-table approach.

In this paper, we assume an organization in which there is a *single table* containing both current and historical data. Commercial systems that implement this technique include Oracle 11g where the concept is called "version-enabled tables" [48]. IBM's DB2 also uses this approach for tables whose only temporal attribute is the application time. The single-table approach is central to IBM DB2 with BLU Acceleration as well [3]. New rows are appended to the table, so the entire table is clustered by begin-time. Updates need to read the table once and write a new version of the record to the end of the table.

We focus on applications that primarily use current data, but occasionally need to access older versions of the data. To support queries over current data, the most recent data may be extensively indexed. Older data may be less heavily indexed because it is queried less frequently and is often more voluminous. Even within a single table, the system can offer an implementation option in which only the most recent version of a record appears in an index.

2 Basic indirection structure

Traditional index structures directly reference a record via a pointer known as a physical row-identifier (RID). The RID usually encodes a combination of the database partition identifier, the page number within the partition, and the row number within the page. A RID index over current HDD-resident data is shown in Fig. 2.

The choice of a physical identifier hinders the update performance of a multiversion database in which updates result

[2] If one wanted to cluster the main table by a temporal attribute to improve temporal locality, then updates would not be in-place and additional indexes would need to be updated. Our proposed solution would reduce the burden of such index updates.

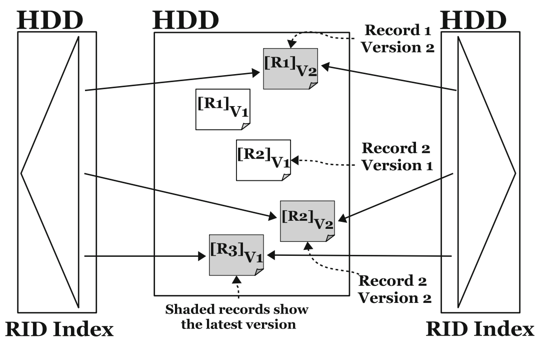


Fig. 2 Traditional RID index structure

in a new physical location for the updated record. Changes to the record induce I/O for every index, even indexes on "unaffected" attributes, i.e., attributes that have not changed. Random I/Os are required to modify HDD-resident leaf pages.

To avoid HDD I/O for indexes on unaffected attributes, we decouple the physical and logical representations of records spanning many versions. We distinguish between a physical row-identifier (RID) and a logical record identifier (LID). For any given record, there may be many RIDs for that record corresponding to the physical placement of all of the versions of that record. In contrast, the LID is a reference to the RID representing the most recent version of the record. For now, one can think of a table *LtoR(LID,RID)* that has LID as the primary key. Indexes now contain LIDs rather than RIDs in their leaves.

Under our proposed Indirection technique, an index traversal must convert a LID to a RID using the *LtoR* table. A missing LID, or a LID with a NULL RID in the *LtoR* table are treated as deleted rows, and are ignored during the search. By placing the *LtoR* table on an SSD, we ensure that the I/O overhead for the extra indirection is relatively small.[3] Because the SSD is persistent, index structures can be recovered after a crash. Because we only need a few SSD bytes per record, it is possible to handle a large magnetic disk footprint with a much smaller solid-state footprint. The new index design is demonstrated in Fig. 3.

When an existing record is modified, a new version of that record is created. The *LtoR* table is updated to associate the new row's RID to the existing LID. That way, indexes on unchanged attributes remain valid. Only for the changed attribute value will index I/O be required for the indirection layer.

When a record is deleted, the (LID,RID) pair for this record in the *LtoR* table is deleted. Index traversals ignore missing LIDs. Indexes can lazily update their leaves during traversal, when a read I/O is performed anyway. At that time,

[3] Frequently accessed LIDs would naturally be cached in RAM by the database bufferpool manager, further reducing the overhead.

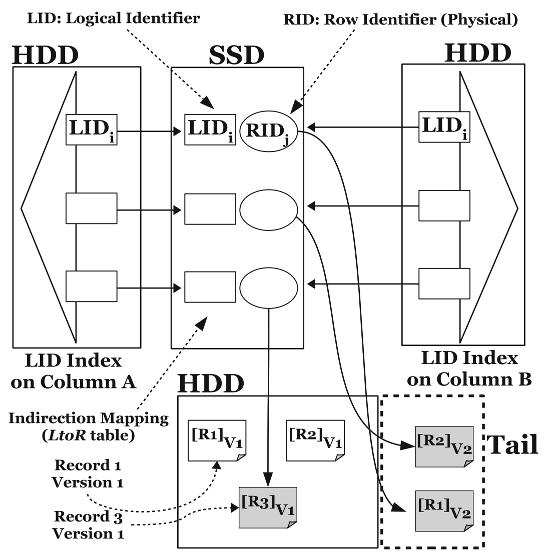


Fig. 3 LID index using the Indirection technique

any missing LIDs encountered lead to the removal of those LIDs from the index leaf page. After a long period of activity, indexes should be validated off-line against the *LtoR* table to remove deleted LIDs that have subsequently never been searched for.

When a new record is added, the new record is appended to the tail of the relation and its RID is fetched and associated with a new LID. The (LID, RID) pair for the new record is added to the *LtoR* table. All indexes are also updated with the new record LID accordingly. In Sect. 3, we discuss how to further improve record insertion and deletion.

3 Enhancing insertions

We now develop techniques for improving the index performance of insertion. We define a batching structure called a LIDBlock, and employ yet another level of indirection.

3.1 LIDBlocks

To reduce the index overhead for insertions, we propose an SSD-resident auxiliary LIDBlock structure containing a fixed number of LIDs. The LIDs in a LIDBlock may be NULL, or may be valid LIDs from the *LtoR* table. References to a LIDBlock are mapped to multiple LIDs through this extra level of indirection. Figure 4 shows the extended structure with LIDBlocks.

The arrow from index leaf pages to LIDBlocks in Fig. 4 could be implemented by keeping a block identifier (BID) within the leaf page. The disadvantage of such a choice is that the leaf node of the index needs to be read from magnetic disk to locate the BID, which requires extra HDD I/O. Instead, we propose to store LIDBlocks within hash tables on the SSD. In

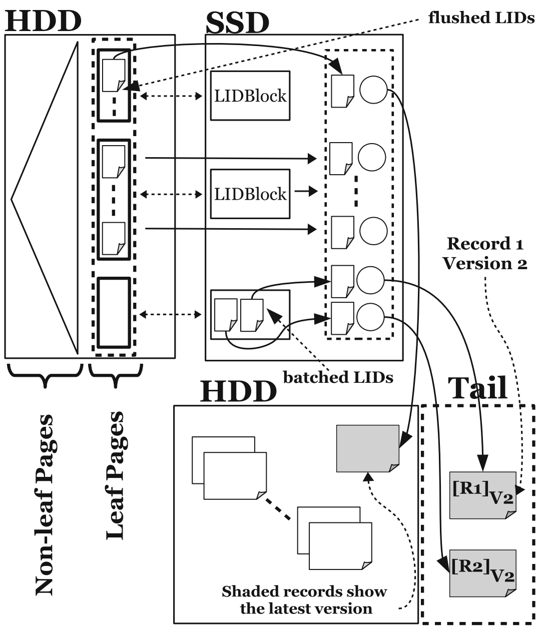


Fig. 4 Indirection technique with LIDBlock

the following sections, we describe more precisely how index pages refer to and traverse LIDBlocks.

3.2 The dense index case

Consider first the case of a dense index, i.e., a secondary index where there are many records (more than a leaf node's worth) for each attribute value. For such indexes, we keep a list of LIDs for each value as before. In addition, we store a collection of LIDBlocks on the SSD in a hash table, hashed by the attribute value. Initially, each indexed value has a LIDBlock whose LIDs are all NULL.

When a new record is inserted, we need to modify the indexes to reflect the insertion. Suppose that the value of an indexed attribute is v, and that the index is dense. A LID is created for the record, and a suitable (LID,RID) pair is added to the *LtoR* table. The LIDBlock B for v is identified by accessing the hash table of LIDBlocks on the SSD. If there are unused (NULL) slots in B, one of the slots is overwritten with the LID of the new record. If there are no unused slots, then all LIDs in B and the LID of the new record are moved in bulk into the LID list in the index, amortizing the I/O cost.[4]

In this model, index traversal is slightly more complex: all the non-NULL LIDs in the LIDBlock for a value also need to be treated as matches. Deletions and attribute value updates may also need to traverse and modify a LIDBlock. There is some additional I/O, but only on solid-state storage.

[4] One can shift the burden of LIDBlock flushing outside of running transactions by triggering an asychronous flush once a LIDBlock becomes full (or nearly full), rather than waiting until it overflows.

3.3 The sparse index case

When there are a few matches per index value, the organization above would need a very large number of LIDBlocks, most of which would be underutilized. Instead, for sparse indexes we maintain a single LIDBlock for an entire leaf node of the index. Rather than using a hash table hashed by attribute value, we use a hash table hashed by the address of the index leaf page. This address can be obtained using a partial traversal of the index, without accessing the leaf node itself. Since the internal nodes of a tree index occupy much less space than the leaf nodes, they are much more likely to be resident in the main memory bufferpool.

Searches have some additional overhead because the shared LIDBlock would need to be consulted even for records that may not match the search condition. There would also be overhead during node splits and merges to maintain the LIDBlock structure.

The overhead of LIDBlocks on searches may be high for sparse indexes. For example, a unique index search would previously only have to look up one main file record. With a LIDBlock for a given key range, a search may need to read b of them, where b is the LIDBlock capacity. This example suggests an optimization: store both the LID and the key in the LIDBlock for sparse indexes. This optimization reduces the capacity of LIDBlocks, but significantly improves the magnetic disk I/O for narrow searches.

3.4 The hybrid index case

For certain workloads, it is not always possible to determine the density and sparsity of an index in advance; furthermore, indexes may be neither strictly dense or sparse. Therefore, it is important to support dynamic adaptation of our proposed LIDBlock structure to index a key space that is partially dense and partially sparse, in which the key distribution also changes over time.

To unify the LIDBlock design for the dense and sparse indexes, we propose to distinguish between key sparsity and density at the granularity of leaf pages as opposed to the granularity of the index structure as a whole. For example, any leaf page can either be assigned as dense or sparse. If a leaf page is marked as dense, then for each key within the leaf page, the key's LIDBlock is retrieved by hashing the key value (i.e., the indexed attribute value), whereas if a leaf page is marked as sparse, then all the keys within the leaf page are associated to a LIDBlock that is retrieved by hashing the address of the leaf page instead.

The ability to make a sparsity versus density decisions at the leaf level enables us to dynamically adapt the LIDBlock organization to accommodate ongoing index changes through gradual local restructuring. Essentially, we will have a hybrid LIDBlock structure such that a different LIDBlock allocation method is used depending on whether a given key range is dense or sparse.

3.5 Revisiting deletions and updates

With LIDBlocks, it is now possible that an update or deletion occurs to a record whose LID is in a LIDBlock rather than the LID list. For deletions, one can simply set the LID to NULL in the LIDBlock. For updates that change the position of the record in the index, one needs to nullify the LID in the previous LIDBlock or LID list, and insert it into the new LIDBlock.

In this way, LIDBlocks also improve the I/O behavior of updates to indexes on changed attributes. In the original scheme, HDD-resident index leaf pages appropriate to the new attribute value needed to be updated with the LID of the updated record. In the enhanced scheme, writes are needed only on SSD-resident LIDBlocks most of the time. Magnetic disk writes are amortized over the number of records per LIDBlock.

4 Delta compression

Our proposed Indirection technique limits the impacts of updates to only indexes defined on columns changed by update transactions. However, the Indirection alone does not address the problem of how to efficiently store the different versions of each record due to frequent updates. As a result, the database size may grow at a much faster rate. To improve the space utilization, we introduce a family of DeltaBlock techniques that exploit the overlap among different versions of the same record. The key observation in managing and storing immutable versions of a record is that there tends to be a large overlap between consecutive versions of every record. Typically only a small set of attributes is changed in transaction processing workloads (e.g., on average only a few columns are changed in the standard TPC benchmarks).

The existing multiversion databases fail to exploit this overlap and redundantly store the unchanged portion of a record or rely on expensive offline compression of the data (not adequate in an online setting) [3,48]. The lack of proactive compression results in the growth of database size at a much faster rate compared to the traditional single-version databases that update in-place and discard the older versions. In the worst case scenario, the database size may grow linearly with the average number of versions for each record.

The general principle of our DeltaBlock techniques is to first accumulate recent updates on the fast SSD-resident structure and to periodically flush these deltas together with the latest version of the record to HDDs. Our DeltaBlock techniques enable an effective compression of records across many versions by reducing the need to redundantly store the unchanged portion of the record for every single update. More importantly, the compression is achieved without sacrificing the query performance.

In what follows, we present four variations of our DeltaBlock technique to achieve fast retrieval and effective compression by trading off flexibility (i.e., update patterns and space allocation) versus performance (i.e., the number of random I/Os). We analyze each DeltaBlock variation with respect to the flexibility and performance dimensions.

4.1 Direct DeltaBlock technique

In our Direct DeltaBlock approach, we introduce a natural and fast data compression and decompression that exploits the overlap among consecutive versions of a record and utilizes non-volatile memory with the fast random access property. The key design feature of Direct DeltaBlock is to avoid changing the magnetic disk I/O pattern for both query and update processing. We facilitate this I/O property by ensuring that the latest version of any record is retrievable with at most one HDD disk I/O and one (or a constant number of) SSD I/Os.[5] In fact, we show that one can do even better and query the latest k versions of a record with at most one disk I/O, a query that otherwise could have required up to k I/Os in traditional multiversion databases.

For each record identified by a LID, we batch on the SSD-resident DeltaBlock structure, a set of deltas between consecutive versions of a record in a given pre-determined DeltaBlock size.[6] Periodically, we flush a set of deltas for each record to disk once its corresponding DeltaBlock overflows, as shown in Fig. 5. More importantly, whenever deltas are flushed to disk, we also reconstruct the latest version of the records given the recent set of deltas, and we simultaneously flush both the deltas and the latest reconstructed version of the record to disk. The flushing of a periodically reconstructed version of the record fulfills our key design principle of retrieving any record with one HDD I/O at most. Also, storing the deltas together with the latest version of the record satisfies the second property of retrieving the latest k versions of the record (or up to $2k$ versions assuming that k deltas could also be accumulated in the DeltaBlock region) with at most one disk I/O. After reconstructing and flushing the latest version of a record, the *LtoR* table is also updated to point to the RID of the most recent version of the record. Notably, the *LtoR* table is now updated much less frequently, only once for every k updates (assuming a DeltaBlock can hold up to k deltas).

[5] Notably, the cost of additional SSD I/Os is negligible with respect to the cost of HDD I/O.

[6] A single DeltaBlock may hold several small updates (e.g., updating only one column at a time) or may hold a single large update (e.g., updating many columns in one transaction).

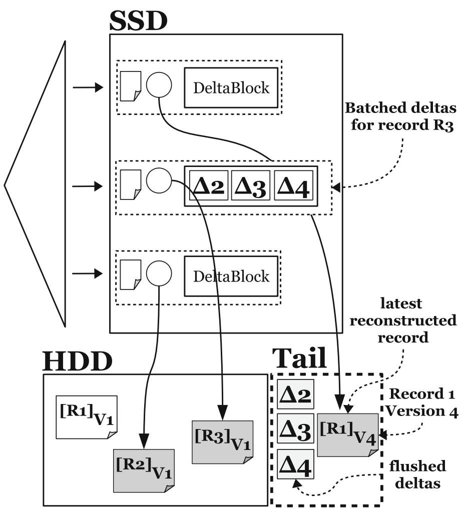


Fig. 5 Direct DeltaBlock technique

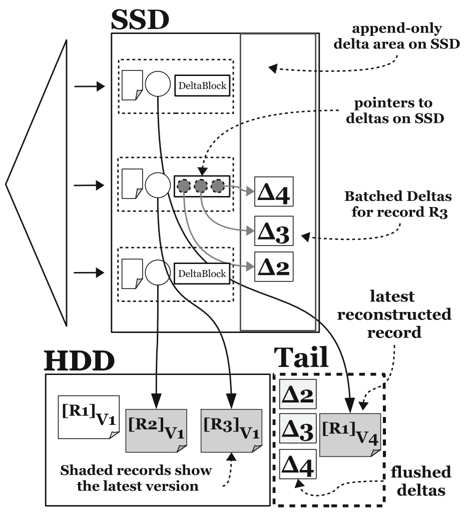

SSD
append-only delta area on SSD
DeltaBlock
pointers to deltas on SSD
Δ4
Δ3
Δ2
Batched Deltas for record R3
DeltaBlock
latest reconstructed record
HDD
Tail
[R1]V1
[R2]V1
[R3]V1
Shaded records show the latest version
Δ2
Δ3
Δ4
[R1]V4
flushed deltas

Fig. 6 Indirect DeltaBlock technique

One way to store these DeltaBlocks is to extend *LtoR* (LID,RID) to *LtoR*(LID, RID, DeltaBlock). Alternatively, DeltaBlock can be implemented as a separate table or a hash table structure. The integration of DeltaBlock by extending the Indirection technique is pictorially presented in Fig. 5, including the batching and flushing of deltas and reconstructing the latest version of a record.

Direct DeltaBlock performs well because the reading/ writing of the latest delta requires at most one random I/O on SSDs. However, without prior knowledge of the workload, it is non-trivial to decide about the initial size of each DeltaBlock; therefore, initially the DeltaBlock size may be over-provisioned. It is important to note that when data are first loaded (or inserted), it is not necessary to allocate any DeltaBlocks for the new records if DeltaBlocks are stored in a secondary structure (such as a hash table) not included in the *LtoR* mapping. DeltaBlocks can be allocated on-demand, and subsequently the size of DeltaBlocks can be tuned to adapt to workload characteristics.

4.2 Indirect DeltaBlock technique

Our second DeltaBlock technique, which we refer to as Indirect DeltaBlock, changes how deltas are stored. Instead of extending the *LtoR* table with sufficient space to hold up to k deltas, we allocate sufficient space in DeltaBlocks to hold only pointers to the actual deltas on the SSD (as shown in Fig. 6). The actual deltas for all records are simply maintained in an append-only fashion; thus, promoting fast sequential writes on SSDs and simplifying the task of determining the DeltaBlock size. In Indirect DeltaBlock, flushing of data from SSDs to HDDs is identical to our Direct DeltaBlock approach.

Under this new scheme, the DeltaBlock size (which could be tuned independently for each record) is proportional to the expected number of deltas, a fixed pointer size for each delta. Although in Indirect DeltaBlock we are required to know the expected number of updates per record (to avoid over-provisioning), we are not required to know the size of each update. Thus, Indirect DeltaBlock is more flexible compared to Direct DeltaBlock. However, with respect to the performance, in Indirect DeltaBlock, the reads are still limited to one random I/O at most on HDDs (to read the last version of the record on disk), and reads are limited to $k + 1$ random I/Os at most on SSDs: one I/O to read the pointers stored in the DeltaBlock and k I/Os to follow the k pointers in the DeltaBlock in order to fetch the values of the k deltas.

Fetching the last k deltas (k random I/Os on SSDs) can further be improved if the deltas are stored cumulatively, in the sense that each delta carries changes from the previous deltas. The accumulation is reset every time DeltaBlock is flushed to hard disk. Thus, in the worst case scenario, k deltas are accumulated, but all k deltas can be fetched by a single random I/O on SSDs.

4.3 Indirect Chained DeltaBlock technique

Our next DeltaBlock technique (called Indirect Chained DeltaBlock) further improves the flexibility dimension by eliminating the need to predict the average number of updates per record. To achieve this end, we propose to allocate at most one pointer in each DeltaBlock to point to the last delta (if any) on SSDs (as shown in Fig. 7).

Similar to the Indirect DeltaBlock technique, deltas are inserted in an append-only fashion on SSDs. However, in Indirect Chained DeltaBlock, we allocate only one pointer in each DeltaBlock. Furthermore, each appended delta also holds a pointer to its previous delta (if any). In this way, the deltas for a single record are chained together, and one is able to follow the pointer in a DeltaBlock to fetch the last delta (the last update) and from the last delta keep following the next

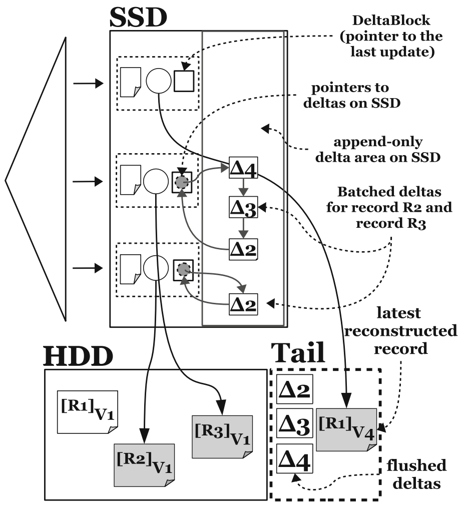

SSD
DeltaBlock (pointer to the last update)
pointers to deltas on SSD
append-only delta area on SSD
Batched deltas for record R2 and record R3
latest reconstructed record
HDD
Tail
[R1]V1
[R2]V1
[R3]V1
[R1]V4
Δ4
Δ3
Δ2
Δ2
flushed deltas

Fig. 7 Indirect Chained DeltaBlock technique

pointer to retrieve all deltas for the corresponding record one-by-one (as shown in Fig. 7). Indirect Chained DeltaBlock is our most flexible technique because it automatically adapts to the workload characteristics; it accommodates both the frequently updated and rarely updated records without the need to predict the average number of updates per record or the size of each update.

In the Indirect DeltaBlock technique, deltas were forced to be flushed as soon as the allocated DeltaBlock for a record was full. However, in Indirect Chained DeltaBlock, the flushing of deltas can be delayed until the entire dedicated append-only delta area is full. We perform a backward pass of the last deltas on SSDs and follow their pointers to retrieve all the deltas for their corresponding records. Once all deltas have been gathered (these fetched deltas are considered as visited), we reach the record's DeltaBlock from which we access the latest version of the record on disk by following the record's RID. Now we are in a position to reconstruct the latest version of the record, and we flush the latest version and all its deltas from the SSD to the HDD. We continue the backward pass and choose the second last unprocessed delta on the SSD until all the deltas are processed.

The latest versions of the record are also flushed to disk in an append-only fashion in the reverse order; hence, after flushing all deltas, the last record on disk is the record that received its first update (first delta) after all other records had been updated at least once.

Similar to the basic Indirect DeltaBlock technique, our proposed Indirect Chained DeltaBlock requires a fixed DeltaBlock size, which is sufficient to store one pointer at most. This choice minimizes the need to over-provision the space. Using the Indirect Chained DeltaBlock, the reads are still limited to at most one random I/O on HDDs and limited to at most a constant number of SSD I/Os (maximum of k I/Os, in which k is the number of deltas that must be chased). The delta accumulation optimization discussed earlier also applies to Indirect Chained DeltaBlock.

4.4 Indirect Chained DeltaBlock technique with deferred compression

In our final DeltaBlock technique, we introduce a deferred compression mechanism in order to eliminate the need to chase deltas during read operations. We call this approach Indirect Chained DeltaBlock with Deferred Compression. We require that the latest constructed version of the record is always available on HDD pointed to by indirection layer (as shown in Fig. 8), which enables a single HDD I/O and zero SSD I/O to reconstruct the latest version of the record. Indirect Chained DeltaBlock enables a fast scan without consulting SSDs for DeltaBlocks.

We divide the tail of the table on HDD into two parts: the compressed and uncompressed portions. In the uncompressed part, we temporarily store the latest version of the recently updated records. The individual deltas for recently updated records (those records that are placed in the uncompressed tail of the table) are stored in an append-only delta area on the SSD. The deltas on the SSD are maintained similarly to the Indirect Chained DeltaBlock technique, but the flushing and subsequently the compression mechanism are different.

Once the append-only delta area on the SSD is full, then the uncompressed tail of the table on the SSD is compressed, and the latest versions of the record together with their deltas are flushed to the hard disk in an append-only fashion (but in the reverse order). However, unlike in Indirect Chained DeltaBlock, the latest versions are already available on the uncompressed tail and no reconstruction is necessary. Thus, by concurrently performing a backward pass on both the

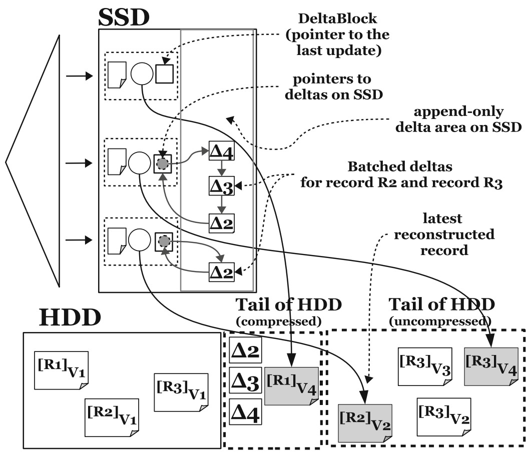

SSD
DeltaBlock (pointer to the last update)
pointers to deltas on SSD
append-only delta area on SSD
Batched deltas for record R2 and record R3
latest reconstructed record
HDD
Tail of HDD (compressed)
Tail of HDD (uncompressed)
[R1]V1
[R2]V1
[R3]V1
Δ2
Δ3
Δ4
[R1]V4
[R2]V2
[R3]V3
[R3]V2
[R3]V4
Δ4
Δ3
Δ2
Δ2

Fig. 8 Indirect Chained DeltaBlock technique with deferred compression

HDD and SSD, we can further speed up the deferred compression of the tail on the HDD. It is important to note that the order of deltas on the SSD is identical to the order of records in the uncompressed tail of the table on the HDD, which allows a simple re-use of the already constructed latest versions of records on the HDD.

Indirect Chained DeltaBlock with Deferred Compression performs similarly to the Direct DeltaBlock with respect to the reads, but without the lack of flexibility of Direct DeltaBlock that requires provisioning the size and the number of updates. However, with respect to the writes, Indirect Chained DeltaBlock with Deferred Compression requires one additional write on the HDD. Since all writes are append-only to the tail of the table, the tail is always expected to be cached in memory, and the cost for flushing the tail is amortized over many writes to a page. Hence each individual write to the tail does not necessarily result in a physical write.

5 Extended example

In this section, we provide a detailed example to further illustrate the core of our Indirection proposal and to motivate the analysis in Sect. 6.

Consider a table $R(A, B, C, D)$ with B-tree indexes on all four attributes. A is the primary key and is therefore sparse. B is also sparse, while C and D are dense. R is stored on disk in a versioned fashion. For each row we show the RID of the previous version (if any) of the row; the previous-RID may or may not be explicitly stored. Suppose that at a certain point in time the extension of R includes the rows given below. A flag indicating whether the row has been deleted is also included.

RID	A	B	C	D	Prev-RID	Deleted
345	100	3732	3	5	123	0
367	120	4728	3	6	NULL	0
369	130	2351	2	5	NULL	0
501	100	3732	2	5	345	0

Suppose the *LtoR* table for R is given by

LID	RID
10	367
11	369
13	501

Indexes use LIDs (e.g., 10, 11, 13) rather than RIDs to refer to rows, and only the most recent version is indexed. Given the above database, the immediate and deferred changes

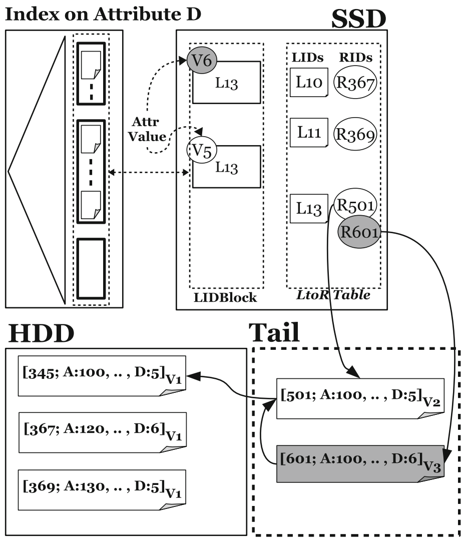

Index on Attribute D
SSD
V6
L13
Attr Value
V5
L13
LIDs
RIDs
L10
R367
L11
R369
L13
R501
R601
LIDBlock
LtoR Table
HDD
Tail
[345; A:100, .. , D:5]$_{V1}$
[367; A:120, .. , D:6]$_{V1}$
[369; A:130, .. , D:5]$_{V1}$
[501; A:100, .. , D:5]$_{V2}$
[601; A:100, .. , D:6]$_{V3}$

Fig. 9 Example of indirection technique with LIDBlock

to the database in response to various update requests are described below. The I/O needed is summarized in square brackets (log I/O and I/O for index node splits/merges are ignored). Our I/O estimates assume that all internal index nodes are resident in the RAM bufferpool and that all leaf nodes require I/O. These estimates also assume that all accesses to SSD-resident structures require I/O.[7] The estimates also assume direct access to a page given the LID, as might be supported by a hash table. We assume that LIDBlocks contain b LIDs, and that for sparse indexes we are storing both the key and the LID in the LIDBlock. Similarly, we assume that a DeltaBlock holds at most k deltas. We now describe precisely each key operation, breaking down its steps into immediate and deferred actions. Immediate actions must be completed within the operation itself. Deferred actions are those that can happen later, such as when an operation causes a page to become dirty in the bufferpool, but the actual I/O write comes later.

Update We update the row with key 100 such that attribute D is changed from 5 to 6. The immediate actions are as follows (also demonstrated in Fig. 9).

- The LID of the row in question (i.e., 13) is identified using the index on A [1 HDD read].
- The entry (13, 501) in the *LtoR* mapping is read [1 SSD read].
- The row with RID 501 is read [1 HDD read].
- A new version of the record is created at the tail of R with a new value for D and a new RID (suppose 601) (if

[7] This might be too pessimistic, particularly for LIDBlocks and DeltaBlocks that could be small enough to be cached in RAM.

DeltaBlock techniques without deferred compression are employed this step can be skipped).
- If the DeltaBlock is employed, then the DeltaBlock entry for the LID 13 is read [1 SSD read].[8]
- An index traversal for key 5 is initiated on attribute D. If LID 13 is present in the corresponding leaf, it is deleted; otherwise, LID 13 is located in the LIDBlock for key 5 and is removed from the LIDBlock [1 HDD read, possibly 1 SSD read].
- A partial index traversal for key 6 is initiated on attribute D, and LID 13 is inserted into the corresponding LIDBlock [1 SSD read].

The required deferred actions are summarized as follows.

- The data page containing the row with RID 601 is dirty and will need to be flushed to the HDD [1 HDD write amortized over all modifications to the page].[9]
- The entry (13, 501) in the *LtoR* mapping is changed to (13, 601) [1 SSD write].
- If the DeltaBlock is employed, then the DeltaBlock entry is changed to hold the updated values directly [1 SSD write]. Alternatively, the updated values are appended to the batched delta area on the SSD, and the pointer in the DeltaBlock is updated to point to the updated values [2 SSD writes].
- The index page containing the key 5 will be dirty if LID 13 was in the leaf. The dirty page will need to be flushed to the HDD [1 HDD write, amortized over all modifications to the page]. If LID 13 was in the LIDBlock for key 5, then the dirty LIDBlock will need to be flushed to the SSD [1 SSD write].
- The LIDBlock for key 6 is dirty and will need to be flushed to the SSD [1 SSD write].

Insertion Consider the insertion of a new record (140, 9278, 2, 6). The resulting immediate actions are as follows.

- The absence of a previous version of the row is verified using the index on A, including the LIDBlock [1 HDD read, 1 SSD read].
- A new row is created at the tail of R, with a new RID (suppose 654). The previous-RID field is NULL.
- A new LID (suppose 15) is allocated.
- For the two dense indexes on C and D, the LIDBlocks for keys 2 and 6 (respectively) are identified, and LID 15 is inserted into each [2 SSD reads].
- For the two sparse indexes on A and B, the LIDBlocks for keys 140 and 9278 (respectively) are identified using partial index traversals, and LID 15 is inserted (paired with the key) into each [2 SSD reads].

The deferred actions are as follows:

- The data page containing the row with RID 654 is dirty and will need to be flushed to the HDD [1 HDD write, amortized over all modifications to the page].
- The entry (15, 654) is inserted into the *LtoR* mapping [1 SSD read, 1 SSD write].
- The LIDBlocks for each of the four indexes are dirty and need to be flushed [4 SSD writes].
- In the event that a LIDBlock fills (one time in b insertions), we need to convert all LIDs in the LIDBlock into regular index LIDs and reset the LIDBlock to an empty state [4/b SSD writes, 3/b HDD reads (the leaf in the index on A has already been read), 4/b HDD writes].

Deletion Now suppose the row with key 100 is deleted, this results in the following immediate actions.

- The LID of the row in question (i.e., 13) is identified using the index on A [1 HD read, possibly 1 SSD read].
- The pair (13, 501) in the *LtoR* table is located [1 SSD read].
- The row with RID 501 is read and the deleted flag is set to 1 [1 HDD read].
- LID 13 is removed from the leaf node of the index on A.

The deferred actions for deleting the row with key 100 are as follows:

- The data page containing the row with RID 501 is dirty and will need to be flushed to the disk [1 HDD write, amortized over all modifications to the page].
- The pair (13, 501) in the *LtoR* table is dropped [1 SSD write].
- The index leaf page for A containing the key 100 is dirty and will need to be flushed to the HDD [1 HDD write amortized over all modifications to the page].
- Whenever one of the other indexes is traversed and LID 13 is reached, LID 13 will be removed from the corresponding LID list [1 extra HDD write to modify the leaf, amortized over all modifications to the page].

Search Suppose that the search returns m matches that all fit in one index leaf page of a sparse index. Since no write is involved, the search only consists of immediate actions.

- Traverse the index [1 HDD read].
- Read the LIDBlock for the leaf node [1 SSD read].

[8] For DeltaBlock techniques with the indirect chaining method, only one pointer is needed to point to the last delta, and this pointer can be included as part of the indirection mapping; thus, no additional read is required.

[9] Amortization is expected to be high on the tail of a table.

Table 1 Base versus Indirection technique analysis

Technique	Type	Immediate SSD	Deferred SSD	Immediate HDD	Deferred HDD
Base	Single-attr. update	0	0	$3 + t$	$\leq 2 + t$
	Insertion	0	0	$1 + t$	$\leq 1 + t$
	Deletion	0	0	$2 + t$	$\leq 1 + t$
	Search Uniq.	0	0	2	0
	Search Mult.	0	0	$\lceil m/I \rceil + m$	0
Indirection	Single-attr. update	2	2	3	≤ 2
	Insertion	$1 + t$	$2 + t + t/b$	1	$\leq 1 + (2t - 1)/b$
	Deletion	1	1	2	$\leq 1 + t$
	Search Uniq.	2	0	2	0
	Search Mult.	$\lceil m/I \rceil + m$	0	$\lceil m/I \rceil + m$	0

For single-attribute updates and deletions, we show the case where the LID was initially in the leaf rather than the LIDBlock. n is the number of attributes, t is the number of B-tree indexes, m is the number of results returned by a search, I is the number of keys in an index leaf node, and b is the maximum number of LIDs in a LIDBlock

- Map LIDs to RIDs [m SSD reads].
- Read all matching records [m HDD reads (assuming an unclustered table)].
- If the DeltaBlock techniques without deferred compression are employed, then for each matching record either read the DeltaBlock that directly holds the updated values [1 SSD read] or read the pointer(s) in the DeltaBlock followed by fetching each corresponding delta from the batched delta area [$\leq 1 + k$ SSD reads].

6 Analysis

We analyze the performance of the Indirection and DeltaBlock methods according to three criteria: (a) time for core operations; (b) SSD space requirements; (c) SSD endurance and recovery.

6.1 Time complexity

Table 1 summarizes the I/O complexity of the Base and Indirection methods, and Table 2 summarizes the I/O complexity of DeltaBlock without the cost of Indirection. Most I/Os are random, meaning that the SSD I/Os will be much faster than HDD I/Os by about two orders of magnitude on current devices. It is therefore worth investing a few extra SSD I/Os to save even one HDD I/O. Note that these estimates are pessimistic in that they assume that none of the SSD-resident data is cached in RAM. If commonly and/or recently accessed data was cached in RAM, many SSD read I/Os could be avoided, and many SSD writes could be combined into fewer I/Os; thus, reducing I/O complexity and improving the device endurance.

The most striking observation is that with a small increase in SSD I/Os, the update, insertion, and deletion costs are substantially reduced, while the cost of searches increases slightly. With the Indirection technique, the immediate HDD cost is independent of the number of indexes. Furthermore, we observe that with a constant increase in the number of SSD I/Os for query, we can substantially reduce the amount of data written to disk using the DeltaBlock techniques, essentially eliminating the need to redundantly write the unchanged portion of the record upon every update.

6.2 SSD space complexity

We now estimate the space needed on the SSD for the *LtoR* table, the LIDBlocks, and the DeltaBlocks. If N is the number of latest version records in a table, then we need N (LID,RID) pairs in the *LtoR* table. In a typical scenario, table records may be 150 bytes wide,[10] with 8-byte LIDs and 8-byte RIDs. Assuming a fudge-factor of 1.2 for representing a hash table, the *LtoR* table would roughly constitute 13 % of the size of the current data in the main table.

We now consider the space consumption of LIDBlocks for sparse indexes; dense indexes would take less space. The number of LIDBlocks for a single sparse index is equal to the number of leaves in the index. With an 8-byte key, an 8-byte LID, an 8 KB HDD page size and a 2/3 occupancy factor, a leaf can represent about 340 records. A LIDBlock contains b (LID,key) pairs, leading to a total space consumption of $16bN \times 1.2/340$ bytes per index. Even a small value of b, say 16, is sufficiently large to effectively amortize insertions. At this value of b, the LIDBlocks for a single index constitute less than 1 % of the size of the current data in the main table.

[10] For example, the average record size in the `LINEITEM` table in TPC-H is around 150 bytes when start/end timestamps for tracking versions are also accounted for [60].

Table 2 DeltaBlock techniques analysis not including the I/O cost incurred by the indirection and LIDBlock techniques as captured in Table 1

Technique	Type	Immediate SSD	Deferred SSD	Deferred HDD
Direct DeltaBlock	Single-attr. update	1	1	0
	Search Uniq.	1	0	0
	Flushing deltas	1	1	1
Indirect DeltaBlock	Single-attr. update	1	2	0
	Search Uniq.	$\leq 1+k$	0	0
	Flushing deltas	$1+k$	$1+k$	1
Indirect Chained DeltaBlock	Single-attr. update	0	2	0
	Search Uniq.	$\leq k$	0	0
	Flushing deltas	$1+k$	$1+k$	1
Indirect Chained DeltaBlock with Deferred Compression	Single-attr. update	0	2	1
	Search Uniq.	0	0	0
	Flushing deltas	$1+k$	$1+k$	1

For single-attribute updates, we assume that each DeltaBlock (with or without chaining) holds at most k deltas for each record and that the DeltaBlock's content is flushed on a per record basis after the accumulation of k deltas, and the flushed deltas are (pseudo) deleted

Thus, even with 7 indexes per table, the SSD space required in a typical scenario is only about 20 % of the HDD space required for the current data. Taking the historical data into account, the relative usage of SSD space would be even lower. Given that SSDs are more expensive per byte than HDDs, it is reassuring that a well-provisioned system would need much less SSD capacity than HDD capacity.

Lastly, we focus on non-direct DeltaBlock techniques, which do not require explicit knowledge about the workload characteristics such as the expected update size per record. For Indirect DeltaBlock, if we assume at most 3 updates per record, then we can extend the Indirection to include three pointers to the batched delta area on the SSD. If we assume 8 bytes per pointer (in most cases, a 4-byte pointer size should be sufficient because the batched delta area is sized to hold only recent updates), then each Indirection entry extended with DeltaBlock will be 40 bytes in the worst case, the case that every record is provisioned to hold 3 pointers. Assuming a fudge-factor of 1.2 for representing a hash table, the *LtoR* table extended with DeltaBlock would constitute roughly 32 % of the size of the current data in the main table (or 23 % if 4-byte pointers are used). Now after applying the chaining idea to DeltaBlock, at most one additional pointer is required per (LID, RID) pair. Therefore, the *LtoR* table extended with DeltaBlock would only constitute 19 % of the size of the current data in the main table (or 16 % if 4-byte pointers are used).

6.3 SSD life-expectancy and recovery

SSDs based on enterprise-grade SLC flash memory are rated for about 10^5 erase cycles before a page wears out [40,62]. SSDs internally implement wear-leveling algorithms that spread the load among all physical pages, so that no individual page wears out early. SSDs based on phase-change memory (PCM) can sustain a higher erase-cycle rate. For example, current PCM prototypes offer 10^6 to 10^8 erase cycles, while future PCM is expected to have an endurance-level of up to 10^{12} [62]. Furthermore, future spin-transfer torque memory has an expected endurance of up to 10^{15} erase cycles [62].

Focusing on today's SSD technologies, the endurance of these devices is also quantified by a simpler Drive Writes Per Day (DWPD) metric for a certain period of time. For instance, a device with a DWPD rating of 10 with a 5-year guarantee translates to writing the entire device capacity 10 times a day for 5 years. The recent enterprise SSDs marketed for OLTP workloads have a DWPD rating of around 30, e.g., Toshiba PX02SSF010 eMLC 100 GB.[11]

In flash-based SSDs, there is a write-amplification phenomenon, in which the internal movement of data to generate new erase units adds to the application write workload. This amplification factor has been estimated at about 2 for common scenarios [25]. This amplification factor is essential when relying on a raw erase-cycle count of the device, but when using the DWPD metric, the write-amplification phenomenon is already accounted for by the device manufacturer. In general, today's SSDs with a high DWPD rating rely heavily on over-provisioning of space as opposed to developing flash technologies that offer higher erase cycles.

To estimate the wear-out threshold, suppose that the Indirection (or the DeltaBlock) method uses an SSD page size of

[11] Toshiba Enterprise SSD (eSSD): http://toshiba.semicon-storage.com/us/product/storage-products/enterprise-ssd/px02ssb-px02ssfxxx.html

512 bytes. Then a device[12] with a capacity of 8×10^7KB can tolerate 8×10^{12} writes before wear-out, assuming a write-amplification of 2 and 10^5 erases per page.

Our most write-intensive operation is insertion, with about $1 + k$ SSD writes per insertion when there are k indexes.[13] Assuming nine indexes and ten insertions per transaction, we would need 100 SSD writes per transaction. Even running continuously at the high rate of 800 transactions/s, the device would last more than 3 years.

Now we consider the same write-intensive workload with an SSD rated for DWPD at 30 for 5 years.[14] Again suppose that the Indirection method uses an SSD page size of 512 bytes. Then a device with a capacity of 8×10^7KB can be written 30 times daily for 5 years, which translates to 24×10^8 KB total writes per day. Thus, the device can sustain 542 transactions/second for 5 years.

The pages in the *LtoR* mapping can be aligned to the database pages and be cached in RAM (i.e., the database bufferpool). Typically OLTP page sizes are 4–8 KB (a smaller page size improves both the bufferpool hit ratio and reduces the page-latch contention) [45,55]. For example, suppose 4 KB page are used, then we can sustain up to 68 transactions/second for 5 years given our write-intensive workloads. But, more importantly, once SSD pages are included in the database bufferpool, then the commonly and/or recently accessed pages will be RAM-resident. As a result, many SSD writes could be combined into fewer physical I/Os, and dirty pages are flushed only periodically depending on the database checkpointing and bufferpool policies; hence, reducing the I/O complexity and improving the device endurance significantly. Therefore, we can revisit our above analysis as follows: if a physical I/O write is performed for each n logical writes due to locality in the bufferpool, then in our write-intensive workload with a 4 KB page size, we would sustain up to $68n$ transactions/second for 5 years.

To protect the *LtoR* mapping[15] against media failure and to leverage caching of SSD pages in RAM (which further requires protection against power failure), the *LtoR* pages must be logged similarly to pages for any other database objects. In the absence of a log, there will be no data loss; however, all indexes must be rebuilt at start time because all indexes contain LIDs (not RIDs) and rely on the *LtoR* mapping.

[12] Device characteristics are based on an 80 GB SLC FusionIO device.

[13] Again, this estimate is pessimistic because it assumes no RAM caching of SSD pages.

[14] We based our calculation on Toshiba PX02SSF010 eMLC 100 GB, but we limit ourselves to only 80 GB for comparable analysis with our FusionIO device.

[15] The recovery of LIDBlocks and DeltaBlocks follows the same principle.

7 Experimental evaluation

We present a comprehensive evaluation of our Indirection prototype based on the generalized search tree (GiST) index package [29,31]. Additionally, we provide complementary kinds of evidence using DB2 [2] in two ways to further support the results demonstrated using our prototype. Since we are targeting operational data stores with both transactions and analytic queries, we base our evaluation on the TPC-H benchmark [60].

First, in Sect. 7.2 we try to answer the question "How would the performance of a state-of-the-art database change if it were to use our methods?" We employ the commercial database system DB2, but other database systems would have been equally good choices. Given a workload W, we construct a rewritten workload W' for the DB2 engine that simulates the I/O behavior of our technique for W. While the workload W' is not identical to W, we argue that the performance of DB2 on W' provides a conservative estimate of the performance of W on a (hypothetical) version of DB2 that implements the Indirection technique.

Second, in Sect. 7.3, we evaluate our Indirection technique by implementing it within the popular GiST index package [29,31]. The GiST package has successfully been deployed into a number of well-known open-source projects including PostgreSQL [6], PostGIS [5], OpenFTS [4], BioPostgres [1], and YAGO2 [7]. All aspects of the method (insertions, deletions, modifications) have been implemented; we refer to the resulting prototype as LIBGiST^{mv}. We profile the I/O behavior of LIBGiST^{mv} and create a detailed I/O and execution cost model for the Indirection technique.

Finally, in Sect. 7.4, we shift our focus to TPC-H style analytical query processing that is geared toward an operational data store. We provide evidence for the key tenet of this work, namely, *reducing the burden of index maintenance means that the system can afford more indexes, which in turn improves the performance of analytical query processing*.

7.1 Platform

The experiments were ran on the machine equipped with a 6-core Intel Xeon X5650 CPU running at 2.67 GHz, having 32 GB of RAM, two magnetic disks (7200 RPM SATA), and one 80 GB Single-Level Cell (SLC) FusionIO solid-state drive.[16]

In our experiments, we used IBM DB2 version 9.7 [2]. We configured DB2 with adequate bufferpool size (warmed up

[16] Our SLC FusionIO can support up to 88,000 I/O operations per second at 50μs latency. While the FusionIO devices are high-end devices that are relatively expensive, we remark that recent SSDs such as the Intel 520 series can store about 500 GB, cost about \$500, and can support 50,000 I/O operations per second at 85μs latency, more than sufficient for the workloads described here.

Table 3 Query rewriting to capture indirection mechanism

	Base	Indirection
Query	`SELECT *`	`SELECT *`
	`FROM LINEITEM L`	`FROM LINEITEM L, LtoR M`
	`WHERE ?`	`WHERE ? AND`
		`L.LID = M.LID`
Update	`SELECT *`	`SELECT *`
	`FROM LINEITEM L`	`FROM LINEITEM L, LtoR M`
	`WHERE ?`	`WHERE ? AND`
		`L.LID = M.LID`
	`INSERT`	`INSERT`
	`INTO LINEITEM`	`INTO LINEITEM`
	`VALUES (?,···,?)`	`VALUES (?,···,?)`
		`UPDATE LtoR`
		`SET RID = ?`
		`WHERE LID = ?`

prior to starting the experiments) to achieve an average 90 % hit ratio on both data and index pages. For the DB2 experiments, we generate a TPC-H database [60] with scale factor 20. File system caching was disabled in all experiments.

For LIBGiST^{mv}, we extended LIBGiST v.1.0 to a multiversion generalized search tree C++ library that supports our Indirection techniques including LIDBlocks and DeltaBlocks.

7.2 DB2 query rewriting experiments

The goal of the query rewriting experiment is to study the I/O pattern for both the unmodified DB2 system ("Base") and the Indirection approach. To evaluate Base for a query Q we simply run Q in the original schema S. To evaluate Indirection, we rewrite Q into another query Q'. Q' is run in a schema S' containing an explicit LtoR table representing the LID-to-RID mapping. Ideally, the LtoR table is physically located on the SSD device; we empirically examine the impact of the location below. In S', base tables are augmented with an additional LID column, where the value of LID is generated randomly. In S, we build as many indexes as desired on the base tables. In S', we build a single index on the attribute selected in the query, typically the primary key of the base table.

7.2.1 Rewriting queries

For queries, the rewriting simply adds the LtoR table to the FROM clause, with a LID equijoin condition in the WHERE clause. An example template of our query rewriting that simulates the indirection mechanism is shown in Table 3. The queries are written over TPC-H LINEITEM table and the indirection table, denoted by LtoR.

To see why the performance of Q' is a conservative estimate of the cost of the Indirection technique, consider two cases for the query in Table 3. In the first case, some selection condition on an indexed attribute (or combination of conditions) is sufficiently selective that an access plan based on an index lookup is used. This case includes a point query specified as an equality condition on a key attribute. The Base plan for Q would include an index traversal and a RID-based lookup in the LINEITEM table. For Q', we will also have an index traversal and a RID-based lookup of LINEITEM, together with a LID-based lookup of the LtoR table. This is precisely the I/O pattern of the Indirection technique.

In the second case, the selection conditions are not very selective. In such a case, the system is likely to scan the base table in answering Q.[17] To answer Q' in such a case requires a join of LINEITEM and LtoR in order to make sure that we only process the most recent versions of each record. This may actually be an overestimate of the cost needed by the Indirection technique because the Indirection technique can also employ a scan without consulting the LtoR table.

7.2.2 Rewriting updates

For updates, an extra UPDATE statement is added to keep the LtoR table current, as illustrated in Table 3. Since we are simulating a multiversion database, the Base method inserts a new row rather than updating an existing row. While the Base method pays the cost of inserting the new row into each index, we are slightly favoring the Base method by ignoring the cost of deleting the old row from the indexes. Depending

[17] We do not include old versions of records for these experiments; in a true multiversion database the base tables would contain some old record versions that would need to be filtered out during the scan, using the valid time attributes.

on the implementation technique used for temporal attributes (Sect. 1.1), there may be additional I/O required to update the temporal attributes of the old row, but we ignore such I/O here.

For updates in which a single-attribute value is modified, the Indirection method incurs just one index update and one update to the LtoR table. At first, it may seem like there is more work done by the Indirection method simulation for the updates of Table 3 since the INSERT statements are the same, and the Indirection method has an extra UPDATE statement. This impression is true only for the case in which there is one base table index in the base schema S. As soon as there are multiple indexes on base tables in S, the cost of the INSERT statement for the Base method exceeds that of the corresponding INSERT in the Indirection method because more indexes need to be updated.

Our profiling of update statements can easily be extended to delete statements, but we omit such profiling because the performance profile would be similar to that for updates. On the other hand, our rewriting does not model the LIDBlock and DeltaBlock techniques, and, thus, cannot fully capture its performance advantages for insertions. The benefits of the LIDBlock and DeltaBlock techniques will be evaluated in Sect. 7.3.

7.2.3 Results

All DB2 measurements are averages over 5 million randomly chosen queries/updates with the exception of our selectivity experiments and analytical queries, in which fewer operations were performed to complete the experiments in a reasonable time. Furthermore, the necessary query rewritings (as described in Sect. 7.2) were performed outside the DB2, and the average cost of query rewriting was less than 1 % of the overall query execution time (rewriting cost was in the sub-millisecond range), which is negligible.

Effect of indexes on execution time Our first result confirms that adding the extra indirection layer has negligible overhead for query processing as shown in Fig. 10. For this experiment, queries are point queries that are accessed by specifying the value of the primary key attribute. The query overhead is negligible because the indirection mapping from LID-to-RID requires only a single additional random SSD I/O, a delay that is orders of magnitude faster than the necessary random magnetic disk I/O for retrieving data pages holding entire records. Figure 10 also shows that the update execution time for the Base technique increases dramatically as the number of indexes is increased. With 16 indexes, Indirection outperforms Base by a factor of 3.6.

Furthermore, we study the cost of update by breaking it down into (1) reading indexes and the old version of the record (2) writing the new version of the updated record and updating the indexes, and (3) updating the indirection mapping (required for Indirection technique only). As mentioned earlier, the cost of updating the indirection is insignificant with respect to the overall update cost, and, as expected, the cost of Indirection and Base are comparable as far as the affected indexes are concerned. The gap between Indirection and Base increases noticeably as soon as the number of unaffected indexes increases; the update breakdown cost is captured in Fig. 11.

The role of the bufferpool Diving deeper into the bufferpool behavior of DB2 reveals that with a modest number of SSD page writes, the number of magnetic disk index writes is substantially reduced, as shown in Fig. 12. This result demonstrates the effectiveness of Indirection in reducing the index

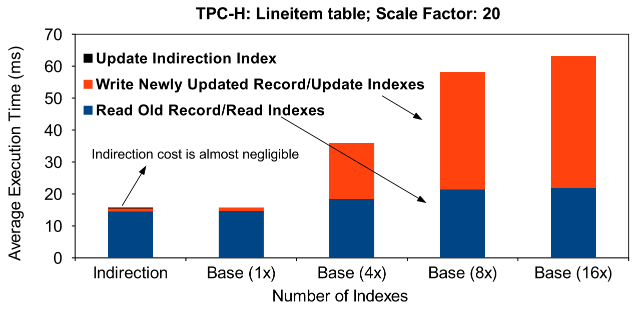


Fig. 11 Varying the number of indexes versus update time

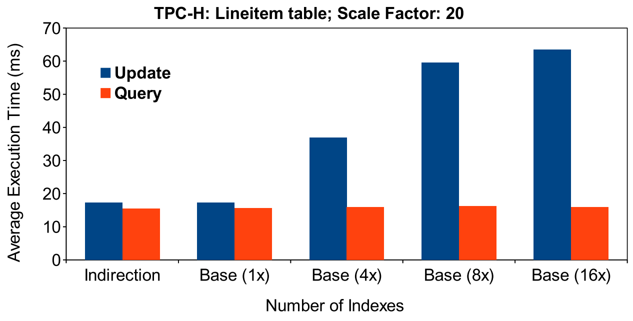


Fig. 10 Varying the number of indexes versus update/query time

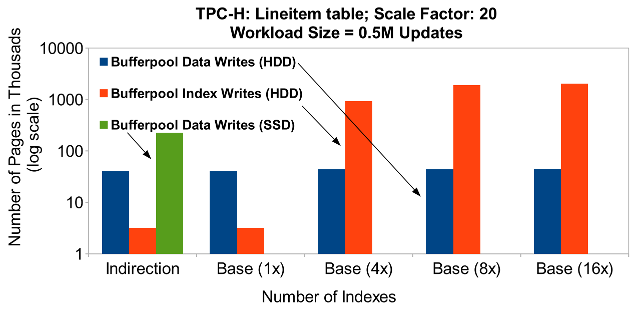


Fig. 12 Varying the number of indexes versus page writes

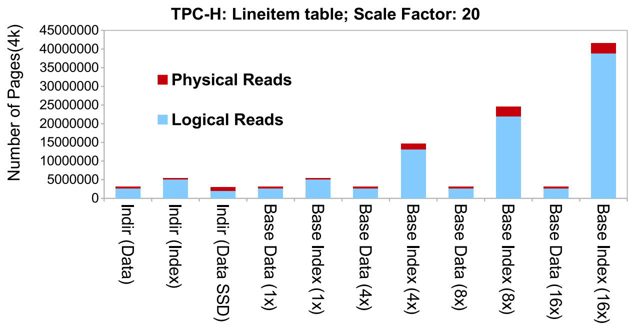


Fig. 13 Bufferpool access breakdown and hit ratio

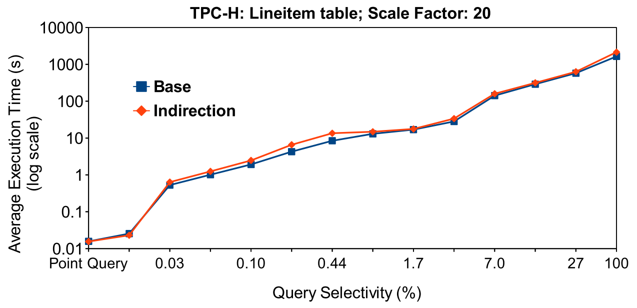


Fig. 14 Varying the query selectivity versus query execution time

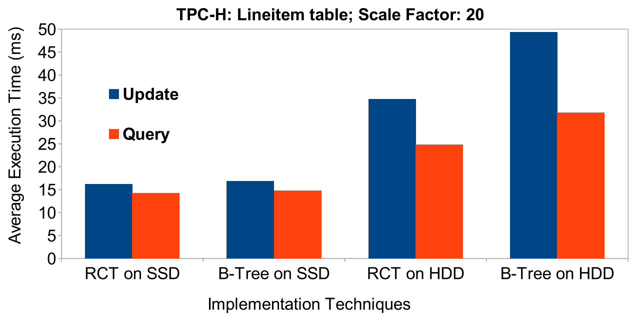


Fig. 15 Varying the indirection implementation techniques

maintenance cost for multiversion databases. Additionally, in all of our DB2 experiments, we ensured that we achieve an average bufferpool hit ratio of around 90 % to resemble real DB2 customer settings [45,55].

We have also carefully studied the different types of accesses served by the bufferpool. In particular, we distinguish among data and index accesses on HDDs versus data access on SSDs (for the indirection mapping). These accesses are furthered classified as logical accesses, i.e., when the accessed page is found and served directly from the bufferpool, and physical accesses, i.e., when the requested page is not found in the bufferpool and a physical access is required to bring the page into memory. As demonstrated in Fig. 13, logical access dominates the total number of accesses, implying a high hit ratio as expected. Furthermore, the majority of accesses to index pages are logical accesses, again as expected, because the index hit ratio are typically even larger for index pages compared to data pages [45,55]. Lastly, we observe the effectiveness of bufferpool for the LtoR table, in which about half of the accesses were served through the bufferpool. We chose a rather conservative/pessimistic hit ratio to ensure fairness. A higher hit ratio not only further improves the Indirection technique execution time, but it further extends the life-expectancy of the SSD device.

Effect of query selectivity and index clustering Consider a range query over a single indexed attribute in the LINEITEM table. By varying the width of the range, we can vary the query selectivity.[18] Figure 14 shows the results for various selectivities, where the indexed attribute is the key of the LINEITEM table by which the table is clustered. On average, the query overhead (for consulting the LtoR table) remains under 20 % as query selectivity varied. There is a sudden jump at around 0.22 % selectivity, but a closer examination of the DB2 query plans reveals that the sudden increase in execution time is attributable to the optimizer incorrectly switching from a nested-loops to a sort-merge join plan.[19]

The 20 % overhead, while still small, is higher than the negligible overhead seen for point queries in Fig. 10. We also observed that as the index clustering ratio decreases, the query processing gap between Indirection and Base decreases. For example, for the lowest clustering ratio index of the LINEITEM table (on the SUPPKEY attribute), the overhead drops to only 4 % on average. These differences can be understood as a consequence of caching. With a high clustering ratio, one base table disk page I/O will be able to satisfy many access requests, meaning that each magnetic disk I/O is amortized over many records. On the other hand, every record will contribute an SSD I/O for the LtoR table since that table is not suitably clustered. For point queries and queries over an unclustered index, the ratio of SSD I/O operations to magnetic disk I/Os will be close to 1.

Indirection mapping implementation Finally, we illustrate that the indirection random read/write access pattern is ideal for SSDs and not for magnetic disks.

We tried two different implementations of the LID-to-RID mapping using either a DB2 range-clustered table (RCT) or a traditional B-Tree index hosted on either SSDs or HDDs. As shown in Fig. 15, when an SSD is used, the overall query and update cost is 1.9× and 2.7× lower, respectively, than on an HDD.

[18] The starting value of the range in our range queries is chosen randomly.

[19] This behavior can be avoided if optimizer hints are provided.

7.3 GiST implementation

Our LIBGiST^{mv} codebase directly implements all aspects of our proposed approach, including the LIDBlock and DeltaBlock techniques. We employ LIBGiST^{mv} as the basis for a systematic performance study in a controlled environment. All HDD and SSD reads and writes used *Direct I/O* to avoid extra copying of operating-system buffers. No bufferpool space was allocated for the indirection table, so requests to the indirection table always incur SSD I/Os.

In our prototype, we also extended the LIBGiST bufferpool in order to test a variety of memory configurations. We further enhanced the LIBGiST library to collect statistics on index operations, file operations, and a detailed bufferpool snapshots. All prototype experiments used the TPC-H schema with scale factor 1, and the workload consisted of random point queries and random insert and update queries (conceptually similar to the workload presented in Sect. 7.2).

We focus on the I/O profile of index traversals and updates as encountered by queries, updates, and insertions. All results are averaged over 10^5 random queries/updates/insertions.

Comparison of average execution time We first isolate the query, update, and insert execution times for a single index with a *small bufferpool size*, enough pages to pin the root of the index and all pages in one path from the root to a leaf. Figure 16 shows that the insert and query times are virtually the same due to the negligible overhead introduced by the additional SSD read and write I/O operations. The "Update" column for the Indirection method in Fig. 16 reflects a traversal of the index to locate the prior version of the record, plus an update of the SSD-resident LID-to-RID mapping. The Base method is more expensive because it has to perform additional writes to the magnetic disk.

Multiple indexes The update time shown in Fig. 16 does not capture the true benefit of the Indirection method when there are multiple indexes. In such a case, the Indirection method needs to traverse only one index, and update one LID-to-RID mapping.[20] In contrast, the Base method needs to traverse and update HDD-resident pages for every index. Figure 17 shows that the performance improvement approaches 20× as the number of indexes increases from 1 to 16.

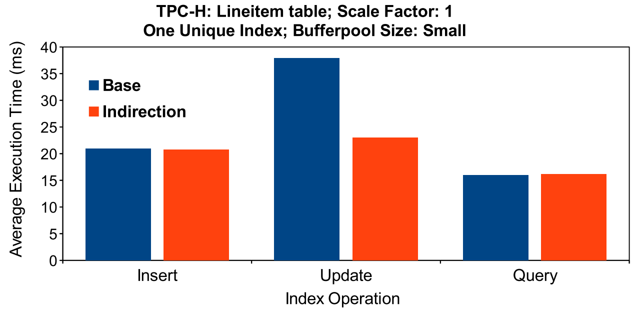


Fig. 16 Insert/Update/Query execution time

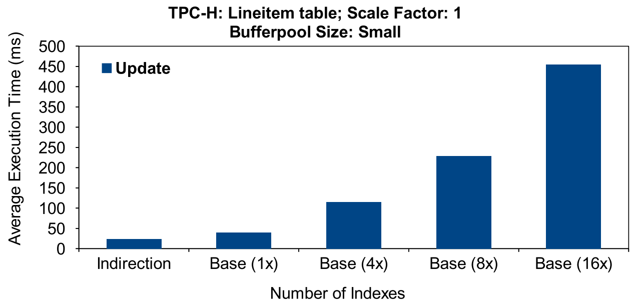
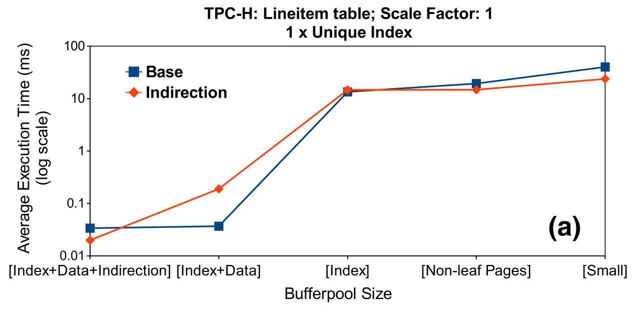


Fig. 17 Varying the number of indexes versus update time



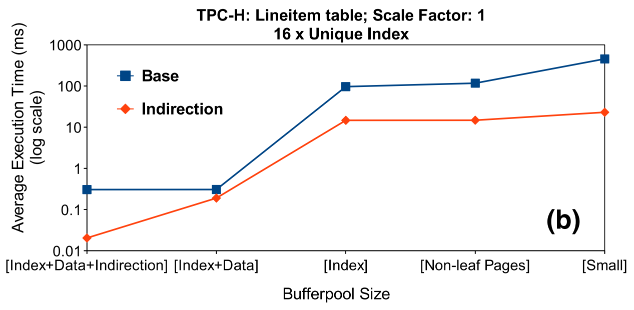


Fig. 18 Varying the bufferpool size versus update time. **a** One index. **b** Sixteen indexes

Varying the bufferpool size We consider five categories of bufferpool sizes large enough to hold: (1) only few pages (small), (2) all index non-leaf pages, (3) the entire index, (4) the entire index and all data pages, or (5) everything including the index, data, and *LtoR* table. These sizes reflect possible use cases where the system has memory ranging from very small to very large. We explored an update workload under these five settings when having one (Fig. 18a) or sixteen (Fig. 18b) indexes on unaffected attributes. Note the logarithmic vertical scale.

[20] If an indexed attribute is updated, then extra I/O is needed for that index to relocate the index entry. Indexes that are neither traversed nor updated in the Indirection method are said to be "unaffected."

In Fig. 18a, when only both the index and data pages are memory-resident, but the LtoR table is not, does the Indirection method perform poorly compared to the Base approach. This is not surprising, because even a fast SSD I/O is slower than access to RAM. When an update operation results in some HDD I/O (either due to updating leaf pages or data pages), then Indirection is superior by up to 2×. When everything including the LID-to-RID mapping table is memory-resident, then Indirection continues to be superior because it does not touch most of the indexes.

Again, the true benefit of the Indirection technique surfaces as we increase the number of indexes. For example, when scaling the number of indexes to sixteen in Fig. 18b, Indirection typically wins by an order of magnitude. These experiments demonstrate that the Indirection technique provides significant benefits even if large portions of the database are memory-resident.

Varying the LIDBlock size So far our focus has been on improving index maintenance involving updates. We now demonstrate the power of our LIDBlock approach for insertions. In Fig. 19, we vary the capacity of LIDBlock from no LIDs to 256 LIDs for the non-unique index defined on the *suppkey* attribute of the *lineitem* table. When increasing the LIDBlock size to around 32 LIDs, we observed that the insertion cost is significantly reduced by up to 15.2×. This improvement is due to the amortization of a random leaf page update over many insertions, e.g., a LIDBlock size of 32 results in batching and flushing to the disk once every 32 insertions on average.

We can demonstrate a similar benefit with a non-unique index having many more duplicate entries, such as an index on the *quantity* attribute of the *lineitem* table, having 50 distinct values and 0.2 M records per value. It is beneficial to allow larger LIDBlock sizes as shown in Fig. 20, in which the insertion execution time is reduced by up to 4.9×. Unlike the previous case, the main reason for the speedup is not simple amortization of insertions; since there are so few key values the tree structure of the index is shallow and its traversal is already cheap due to caching. Instead, the speedup is due to having batches large enough to be resolved with one or two I/Os to update the LID lists for a key.

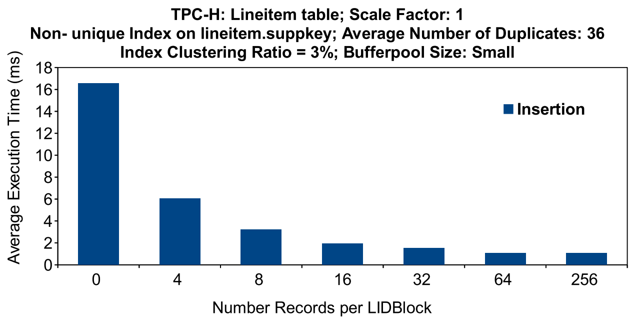


Fig. 19 Varying the LIDBlock size versus insertion time

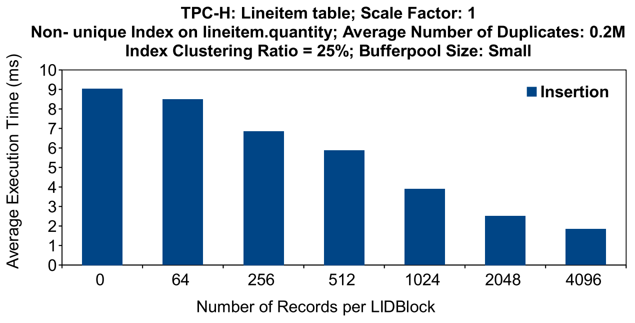


Fig. 20 Varying the LIDBlock size versus insertion time

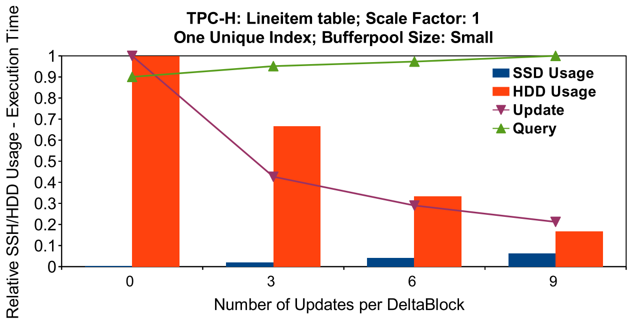


Fig. 21 Varying the DeltaBlock size versus update time

Effect of varying DeltaBlock size Our final prototype experiment demonstrates the role of our Indirect Chained DeltaBlock technique.[21] In this experiment, we begin with one index defined over *lineitem* table (with scale factor 1) followed by randomly updating 100 k randomly selected records up to 12 times. As we scale the number of deltas within a DeltaBlock from 0 to 9, we compute the HDD usage (raw data size), SSD usage (DeltaBlock size), and relative query and update execution time.

As shown in Fig. 21, when increasing the DeltaBlock size, we obtained a compression reduction ratio of 80 % with as few as 9 deltas per DeltaBlock. This reduction is made possible by allocating to DeltaBlock only 6 % of the size of the uncompressed raw data, exhibiting an SSD usage that is only 6 % of HDD usage. Therefore, by maintaining (and chaining) only the values of changed columns (as opposed to redundantly storing all the unchanged columns), we can substantially reduce the size of the database while retaining all past versions.

Since the recent updates are written only to the fast storage (i.e., SSDs) and only the changed columns (deltas) are written, the latest values of frequently updated columns can

[21] Among our DeltaBlock techniques, Indirect Chained DeltaBlock provides the best balance between performance and flexibility. Thus, we consider it as our DeltaBlock representative approach for our empirical studies.

be found on SSDs (i.e., the fetching of the latest values is improved); thereby, the overall update execution time is improved. Moreover, the query time is almost unaffected because the latest updated values can quickly be fetched from SSDs (or the latest version is fetched without the need of reconstruction if the deferred compression technique is employed). In addition, using the DeltaBlock, one can retrieve the latest k versions of record with a single HDD I/O as oppose to the k random I/Os required by the Base technique.

7.4 DB2 operational data store experiments

In this section, we study the effects of adding indexes in the context of an operational data store, in which small-scale analytical query processing is competing with transactional throughput. Our query workload is based on prior work [17] that modifies TPC-H queries so that they each touch less data. For our index workload, we rely on the DB2 Index Advisor recommendation given our query workloads defined over the entire TPC-H schema.

We first consider only the primary key indexes of the TPC-H tables. Subsequently, we add the remaining indexes recommended by DB2 Advisor one at a time, starting from the most to least beneficial index. After each round of index addition, we re-run our query workload. Likewise, after adding each index, we compute the update cost for a uniformly generated update workload, in which each non-primary-key attribute has an equal chance of being updated. The update cost is a normalized average execution time of updating indexes on DB2. The results are summarized in Figs. 22 and 23.

Our first observation is that analytical query time is substantially reduced (by a factor of 3) as we add more indexes recommended by DB2 Advisor. More importantly, we observe that the additional indexes are more "affordable" for updates because our Indirection technique reduces the index maintenance overhead. In the base configuration, the index maintenance overhead increases linearly as more indexes are added, reducing transaction throughput. Our Indirection technique reduces the update cost by more than 40 %.

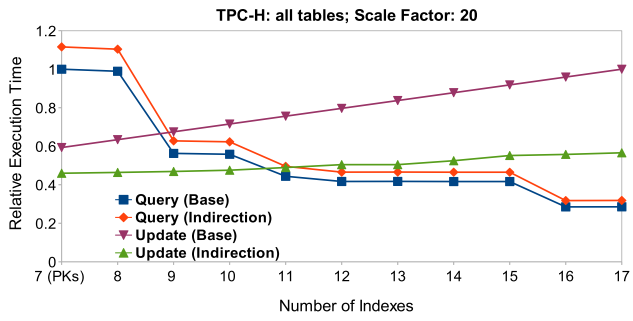


Fig. 22 Effects of adding indexes on query/update time

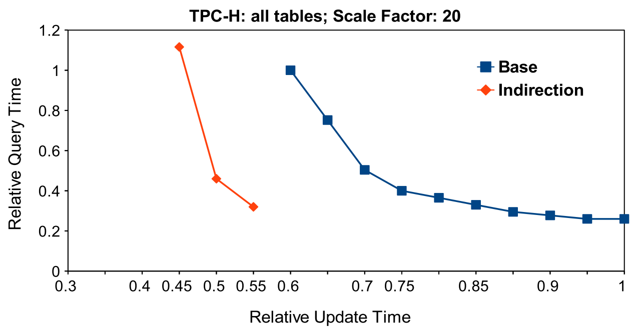


Fig. 23 Effects of adding indexes on query/update time

Figure 23 shows a two-dimensional plot of relative query time q versus relative update time u. On both axes, smaller is better. Each index configuration determines a point (u, q) on this graph, and one can choose a suitable configuration to achieve a desired query/update trade-off. In Fig. 23, the Indirection technique dominates[22] the Base method. This is a key result: The Indirection technique makes indexes more affordable, leading to lower query *and* update times.

To understand the importance of Fig. 23 consider the following scenarios.

1. A DBA has an update time budget of 0.6 units and within that budget wants to optimize query processing time. According to Fig. 23, the Indirection technique can achieve query performance of about 0.32 units under such conditions, while the Base method can achieve only 1.0 units, three times worse.
2. A DBA has a query time budget of 0.5 units and within that budget wants to optimize update time. The Indirection technique can achieve update performance of about 0.5 units under such conditions, while the Base method can achieve only 0.7 units, 40 % worse.
3. A DBA wants to minimize the sum of update time and query time. The Indirection method can achieve a sum of about 0.87 at (0.55, 0.32), whereas the best configuration in the Base method is 1.15 at (0.75, 0.4), 32 % worse.

8 Related work

There has been extensive work on storing high-value data on SSDs. Some of these studies target the problem of data placement in relational databases to take better advantage of the SSD characteristics. In [17] database objects are either placed on HDDs or SSDs based on workload characteristics.[23] As

[22] Except when a very small weight is assigned to update time.

[23] In fact, our indirection mapping can be seen as yet another unique index that is accessed by every query. Thus, the objects placement optimization in [17] can be utilized to determine the globally optimized placement of our indirection object.

opposed to using SSDs and HDDs at the same level in the storage hierarchy, SSDs are also used as a second layer cache between main memory and HDDs [15,18,23,37,44].

The use of an SSD cache with a write-back policy would not solve our problem as effectively as the Indirection technique. One would need space to hold entire indexes if one wants to avoid HDD index I/O for random updates and insertions. In contrast, our method avoids HDD I/O with a much smaller SSD footprint. The Indirection technique and SSD caching of HDD pages are complementary and can be used in tandem. In fact, using Indirection improves the cache behavior by avoiding reading/writing unnecessary pages (i.e., it avoids polluting the cache).

Adding a level of indirection is a commonly used programming technique relevant to a variety of systems problems [10]. The kind of indirection we propose in this paper is used in log-structured file systems [33,51] and database indexing [39], but only at page granularity. We use indirection at record granularity. Furthermore, in [39], a latch-free B-Tree structure, called Bw-Tree, is proposed that virtualizes the concept of the page through page-level indirection (unlike our record-level indirection). But, more importantly, our Indirection technique is index-agnostic (not tuned for any particular index such as B-Tree) and addresses the general problem of index maintenance in multiversion databases, that is, to avoid updating indexes on unaffected attributes. Therefore, our approach could reduce the index maintenance of Bw-Tree as well. Another fundamental difference between our philosophy and Bw-Tree is that we consider de-coupling of the data from the index (in order to simplify the support of secondary indexes on the data), while Bw-Tree assumes a tight coupling of the index and data (i.e., storing the data as part of the index). Bw-Tree is intended as an atomic record store similar to key value stores [39].

The log-structured storage is also exploited in database management systems [41,47,50,57,61], but no indirection layer is used; thus, the common merging and compaction operations in log-structured storage result in expensive rebuilding of all indexes. By employing our Indirection technique this index rebuilding can be avoided.

Page-level indirection tables are also used to improve the lifespan of SSDs. In [21], a system called CAFTL is proposed to eliminate duplicate writes on SSDs. By keeping a mapping table of blocks, the redundant block writes on SSDs are eliminated.

In [63], Wu et al. proposes a software layer called BFTL to store B-tree indexes on flash devices efficiently. IUD operations cause significant byte-wise operations for B-tree reorganization. The proposed layer reduces the performance overhead of these updates on the flash device by aggregating the updates on a particular page.

In [24], Dou et al. propose specialized index structures and algorithms that support querying of historical data in flash-equipped sensor devices. Since the sensor devices have limited memory capacity (SRAM) and the underlying flash devices have certain limitations, there are challenges in maintaining and querying indexes.

The deferral of index I/Os is used in several recent papers on improving index performance [8,14,46]. In those papers, changes are accumulated at intermediate nodes, and propagated to children in batches. Searches need to examine buffers for keys that match. This line of work is complementary but similar to our LIDBlock method in that both techniques buffer insertions to amortize physical I/O.

SSDs are used to support online updates in data warehouses [11]. Incoming updates are first cached in SSDs and later merged with the older records on HDDs to answer queries. In [11], data records are accessed primarily through table scans rather than indexes.

Many specialized indexes for versioned and temporal data have been proposed. A comprehensive survey of temporal indexing methods is provided in [54]. Tree-based indexes on temporal data include the multiversion B-tree [12], Interval B-tree [9], Interval B+-tree [16], TP-Index [58], Append-only Tree [30] and Monotonic B+tree [26]. Efficiently indexing data with branched evolution is discussed by Jouni et al. [36], who build efficient structures to run queries on both current and historical data.

Specialized transaction time database systems, such as Immortal DB [42,43], provide high performance for temporal applications. Lomet et al. [43] describe how a temporal indexing technique, the TSB-tree, is integrated into SQL Server. The paper also describes an efficient page layout for multiversion databases.

Lastly, we have extensively studied the concurrency aspect of our Indirection technique in [52]. By leveraging our Indirection technique, we observed a significant improvement in the execution time of workloads with concurrent read-only and update transactions. Therefore, in the current paper, we have focused exclusively on the benefits of indirection technique as far as the index maintenance and version compression are concerned, while in [52], we have focused exclusively on the role of Indirection technique in concurrency control.

9 Conclusions

The multiversion temporal database market is growing [22]. A temporal database simplifies application development and deployment by pushing the management of temporal logic into database engines. By adopting temporal technologies, the development cost can be reduced by a factor of 10 [22]. This success has led major database vendors (including Oracle [48], IBM [34], and TeraData [59]) to provide support for multiversion temporal data.

We tackle a key challenge of multiversion databases: providing good update performance and good query performance in a single system. Transaction throughput and analytical query processing often have conflicting requirements due to the high index maintenance cost for transactions. Our efficient index maintenance using Indirection makes indexes more "affordable," substantially improving the available configuration choices. Our evaluation demonstrates a query cost reduction by a factor of 3 without an increase in update cost. The batching of insertions using our LIDBlock technique can save up to 90 % of the insertion time. Finally, we demonstrated that our proposed DeltaBlock techniques could substantially reduce the database size by up to 80 % even with a modest space allocation on SSDs.

References

1. BioPostgres: Data management for computational biology. http://www.biopostgres.org/
2. IBM DB2 Database for Linux, UNIX, and Windows. www.ibm.com/software/data/db2/linux-unix-windows/
3. IBM DB2 with BLU Acceleration. www.ibm.com/software/data/db2/linux-unix-windows/db2-blu-acceleration/
4. OpenFTS: Open source full text search engine. http://openfts.sourceforge.net/
5. PostGIS: Geographic information systems. http://postgis.refractions.net/
6. PostgreSQL: Open source object-relational database system. http://www.postgresql.org/
7. YAGO2: High-quality knowledge base. http://www.mpi-inf.mpg.de/yago-naga/yago/
8. Agrawal, D., Ganesan, D., Sitaraman, R.K., Diao, Y., Singh, S.: Lazy-adaptive tree: an optimized index structure for flash devices. PVLDB **2**(1), 361–372 (2009)
9. Ang, C.-H., Tan, K.-P.: The interval B-tree. Inf. Process. Lett. **53**(2), 85–89 (1995)
10. Arpaci-Dusseau, R., Arpaci-Dusseau, A.: Operating Systems: Three Easy Pieces. Arpaci-Dusseau Books, 0.5 edition (2012)
11. Athanassoulis, M., Chen, S., Ailamaki, A., Gibbons, P.B., Stoica, R.: MaSM: efficient online updates in data warehouses. In: SIGMOD Conference, pp. 865–876 (2011)
12. Becker, B., Gschwind, S., Ohler, T., Seeger, B., Widmayer, P.: An asymptotically optimal multiversion B-Tree. VLDB J. **5**(4), 264–275 (1996)
13. Bhattacharjee, B., Lim, L., Malkemus, T., Mihaila, G., Ross, K., Lau, S., McArthur, C., Toth, Z., Sherkat, R.: Efficient index compression in DB2 LUW. Proc. VLDB Endow. **2**(2), 1462–1473 (2009)
14. Bhattacharjee, B., Malkemus, T., Lau, S., Mckeough, S., Kirton, J.-A., Boeschoten, R.V., Kennedy, J.: Efficient bulk deletes for multi dimensionally clustered tables in DB2. In: VLDB, pp. 1197–1206 (2007)
15. Bhattacharjee, B., Ross, K.A., Lang, C.A., Mihaila, G.A., Banikazemi, M.: Enhancing recovery using an SSD buffer pool extension. In: DaMoN, pp. 10–16 (2011)
16. Bozkaya, T., Özsoyoğlu, M.: Indexing valid time intervals. Lect. Notes Comput. Sci. **1460**, 541–550 (1998)
17. Canim, M., Bhattacharjee, B., Mihaila, G.A., Lang, C.A., Ross, K.A.: An object placement advisor for DB2 using solid state storage. PVLDB **2**(2), 1318–1329 (2009)
18. Canim, M., Mihaila, G.A., Bhattacharjee, B., Ross, K.A., Lang, C.A.: SSD bufferpool extensions for database systems. PVLDB **3**(2), 1435–1446 (2010)
19. Chaudhuri, S., Narasayya, V.: Automating statistics management for query optimizers. IEEE Trans. Knowl. Data Eng **13**(1), 7–20 (2001)
20. Chaudhuri, S., Narasayya, V.R.: An efficient cost-driven index selection tool for microsoft SQL server. In: Proceedings of the 23rd International Conference on Very Large Data Bases, VLDB '97, pp. 146–155. Morgan Kaufmann Publishers Inc., San Francisco (1997)
21. Chen, F., Luo, T., Zhang, X.: CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In: FAST, pp. 77–90 (2011)
22. Chen, S.: Time travel query or bi-temporal. In: DB2 for z/OS Technical Forum (2010)
23. Do, J., Zhang, D., Patel, J.M., DeWitt, D.J., Naughton, J.F., Halverson, A.: Turbocharging DBMS buffer pool using SSDs. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, SIGMOD '11, pp. 1113–1124. ACM, New York (2011)
24. Dou, A.J., Lin, S., Kalogeraki, V.: Real-time querying of historical data in flash-equipped sensor devices. In: IEEE Real-Time Systems Symposium, pp. 335–344 (2008)
25. Drossel, G.: Methodologies for calculating SSD usable life. In: Storage Developer Conference (2009)
26. Elmasri, R., Wuu, G.T.J., Kouramajian, V.: The time index and the monotonic B+-tree. In: Temporal Databases, pp. 433–456 (1993)
27. Fusion-io breaks one billion IOPS barrier. http://www.fusionio.com/press-releases/fusion-io-breaks-one-billion-iops-barrier/
28. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems: The Complete Book, 2nd edn. Prentice Hall Press, Upper Saddle River, NJ (2008)
29. The GiST indexing project. http://gist.cs.berkeley.edu/
30. Gunadhi, H., Segev, A.: Efficient indexing methods for temporal relations. IEEE Trans. Knowl. Data Eng. **5**(3), 496 (1993)
31. Hellerstein, J.M., Naughton, J.F., Pfeffer, A.: Generalized search trees for database systems. In: Proceedings of the 21th International Conference on Very Large Data Bases, VLDB '95, pp. 562–573. Morgan Kaufmann Publishers Inc., San Francisco (1995)
32. Hinshaw, F.D., Harris, C.S., Sarin, S.K.: Controlling visibility in multi-version database systems. US 7305386 Patent, Netezza Corporation (2007)
33. Hitz, D., Lau, J., Malcolm, M.: File system design for an NFS file server appliance. In: Proceedings of the USENIX Winter 1994 Technical Conference, WTEC'94, pp. 19–19. USENIX Association, Berkeley (1994)
34. DB2 10 for z/OS. ftp://public.dhe.ibm.com/software/systemz/whitepapers/DB210_for_zOS_Upgrade_ebook
35. Inmon, W.H.: Building the Operational Data Store, 2nd edn. Wiley, New York (1999)
36. Jouini, K., Jomier, G.: Indexing multiversion databases. In: Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM '07, pp. 915–918. ACM, New York (2007)
37. Kang, W.-H., Lee, S.-W., Moon, B.: Flash-based extended cache for higher throughput and faster recovery. PVLDB **5**(11), 1615–1626 (2012)
38. Larson, P.-A., Blanas, S., Diaconu, C., Freedman, C., Patel, J.M., Zwilling, M.: High-performance concurrency control mechanisms for main-memory databases. Proc. VLDB Endow. **5**(4), 298–309 (2011)
39. Levandoski, J.J., Lomet, D.B., Sengupta, S.: The Bw-Tree: a B-tree for new hardware platforms. In: Proceedings of the 2013 IEEE 29th International Conference on Data Engineering, ICDE '13. IEEE Computer Society, Washington (2013)

40. Leventhal, A.: Flash storage memory. Commun. ACM **51**(7), 47–51 (2008)
41. Li, Y., He, B., Luo, Q., Yi, K.: Tree indexing on flash disks. In: Proceedings of the 2009 IEEE International Conference on Data Engineering, ICDE '09, pp. 1303–1306. IEEE Computer Society, Washington (2009)
42. Lomet, D., Barga, R., Mokbel, M.F., Shegalov, G., Wang, R., Zhu, Y.: Immortal DB: transaction time support for SQL server. In: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, SIGMOD '05, pp. 939–941. ACM, New York (2005)
43. Lomet, D., Hong, M., Nehme, R., Zhang, R.: Transaction time indexing with version compression. Proc. VLDB Endow. **1**(1), 870–881 (2008)
44. Menon, P., Rabl, T., Sadoghi, M., Jacobsen, H.: CaSSanDra: an SSD boosted key-value store. In: IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31–April 4, 2014, pp. 1162–1167 (2014)
45. Murphy, G., Compher, D.: DB2 storage observations (2011)
46. Omiecinski, E., Liu, W., Akyildiz, I.F.: Analysis of a deferred and incremental update strategy for secondary indexes. Inf. Syst. **16**(3), 345–356 (1991)
47. O'Neil, P.E., Cheng, E., Gawlick, D., O'Neil, E.J.: The log-structured merge-tree (LSM-Tree). Acta Inf. **33**(4), 351–385 (1996)
48. Oracle database 11g workspace manager overview. http://www.oracle.com/technetwork/database/twp-appdev-workspace-manager-11g-128289
49. Oracle total recall/flashback data archive. http://www.oracle.com/technetwork/issue-archive/2008/08-jul/flashback-data-archive-whitepaper-129145
50. Rabl, T., Gómez-Villamor, S., Sadoghi, M., Muntés-Mulero, V., Jacobsen, H.-A., Mankovskii, S.: Solving big data challenges for enterprise application performance management. Proc. VLDB Endow. **5**(12), 1724–1735 (2012)
51. Rosenblum, M., Ousterhout, J.K.: The design and implementation of a log-structured file system. ACM Trans. Comput. Syst. **10**(1), 26–52 (1992)
52. Sadoghi, M., Canim, M., Bhattacharjee, B., Nagel, F., Ross, K.A.: Reducing database locking contention through multi-version concurrency. Proc. VLDB Endow. **7**(13), 1331–1342 (2014)
53. Sadoghi, M., Ross, K.A., Canim, M., Bhattacharjee, B.: Making updates disk-I/O friendly using SSDs. Proc. VLDB Endow. **6**(11), 997–1008 (2013)
54. Salzberg and Tsotras: Comparison of access methods for time-evolving data. CSURV. Comput. Surv. **31**(2), 158–221 (1999). doi:10.1145/319806.319816
55. Samy, V., Lu, W., Rada, A., Punit, S., Srinivasan, S.: Best practices physical database design for online transaction processing (OLTP) environments (2011)
56. Saracco, C.M., Nicola, M., Gandhi, L.: A matter of time: temporal data management in DB2 for z/OS (2010)
57. Sears, R., Ramakrishnan, R.: bLSM: a general purpose log structured merge tree. In: SIGMOD Conference, pp. 217–228 (2012)
58. Shen, H., Chin, B., Lu, O.H.: The TP-Index: a dynamic and efficient indexing mechanism for temporal databases. In: Proceedings of the Tenth International Conference on Data Engineering, pp. 274–281. IEEE (1994)
59. Snodgrass, R.T.: A case study of temporal data. Teradata Corporation, Dayton (2010)
60. TPC-H, decision support benchmark. http://www.tpc.org/tpch/
61. Vo, H.T., Wang, S., Agrawal, D., Chen, G., Ooi, B.C.: LogBase: a scalable log-structured database system in the cloud. PVLDB **5**(10), 1004–1015 (2012)
62. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: lightweight persistent memory. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI, pp. 91–104. ACM, New York (2011)
63. Wu, C.-H., Kuo, T.-W., Chang, L.-P.: An efficient B-tree layer implementation for flash-memory storage systems. ACM Trans. Embedded Comput. Syst. **6**(3) (2007). doi:10.1145/1275986.1275991