Multi-Version Databases on Flash: Append Storage and Access Paths

Robert Gottstein
Databases and Distributed Systems Group
TU-Darmstadt, Germany
gottstein@dvs.tu-darmstadt.de

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

Alejandro Buchmann
Databases and Distributed Systems Group
TU-Darmstadt, Germany
buchmann@dvs.tu-darmstadt.de

***Abstract*—New storage technologies, such as Flash and Non-Volatile Memories, with fundamentally different properties are appearing. Leveraging their performance and endurance requires a redesign of existing architecture and algorithms in modern high performance databases. Multi-Version Concurrency Control (MVCC) approaches in database systems, maintain multiple timestamped versions of a tuple. Once a transaction reads a tuple the database system tracks and returns the respective version eliminating lock-requests. Hence, under MVCC reads are never blocked, which leverages well the excellent read performance (high throughput, low latency) of new storage technologies. The read performance is also utilised by the read-intensive visibility and validity rules (MVCC, Snapshot Isolation) that filter the latest committed version of a tuple that a transaction can see out of the set of all tuple versions. Much more critical is the update behaviour of MVCC and Snapshot Isolation (SI) approaches, even though conceptually new versions are separate physical entities, which can be stored out-of-place thus avoiding in-place updates. Upon tuple updates, established implementations lead to multiple random writes – caused by (i) creation of the new and (ii) in-place invalidation of the old version – thus generating suboptimal access patterns for the new storage media. The combination of an append based storage manager operating with tuple granularity and snapshot isolation addresses asymmetry and in-place updates. In this paper, we highlight novel aspects of log-based storage, in multi-version database systems on new storage media. We claim that multi-versioning and append-based storage can be used to effectively address asymmetry and endurance. We identify multi-versioning as the approach to address data-placement in complex memory hierarchies. We focus on: *version handling*, (physical) *version placement*, *compression* and *collocation* of tuple versions on Flash storage and in complex memory hierarchies. We identify possible read- and cache-related optimizations.**

Keywords*—*Multi Version Concurrency Control, Snapshot Isolation, Versioning, Append Storage, Flash, Data Placement, Index.

I. INTRODUCTION

This paper is a follow-up, extended paper to our short paper published at the DBKDA 2013 [1]. We describe our Snapshot Isolation Append Storage algorithm (SIAS – [2]) in more detail, show more results of the comparison to other storage mechanisms and deliver more detailed analysis.

New storage technologies such as flash and non-volatile memories have fundamentally different characteristics compared to traditional storage such as magnetic discs. Performance and endurance of these new storage technologies highly depend on the I/O access patterns.

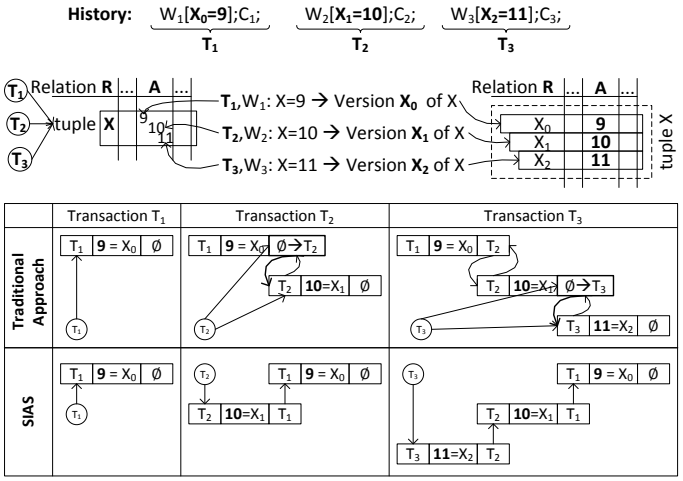


Fig. 1. Invalidation in SI and SIAS

Multi-Version approaches maintaining versions of tuples, effectively leverage some of their properties such as fast reads and low latency. Yet, asymmetry and slow in-place updates need to be addressed on architectural and algorithmic levels of the DBMS. Snapshot Isolation (SI) has been implemented in many commercial and open-source systems: Oracle, IBM DB2, PostgreSQL, Microsoft SQL Server 2005, Berkeley DB, Ingres, etc. In some systems, SI is a separate isolation level, in others used to handle serializable isolation.

Under the concept of *Append-based storage* management any newly written data is appended at the logical head of a circular append log. This way, random writes are eliminated as they get transformed into sequential writes. In-place update operations are reduced to a controlled append of the data, which is an effective mechanism to address the assymmetric performance of new storage technologies (see Section III).

In SIAS [2], we combine the multi-versioning algorithm of snapshot isolation and append storage management (with tuple granularity) on Flash. Under TPC-C workload SIAS achieves up to 4x performance improvement on Flash SSDs, a significant *write overhead reduction* (up to 52x), better *space utilization* due to denser version packing per page, better *I/O parallelism* and up to 4x lower disk I/O execution times, compared to traditional approaches. SIAS aids better *endurance*, due to the use of out-of-place writes as appends and write overhead reduction.

SIAS implicitly invalidates tuple versions by creating a successor version; thus, avoiding in-place updates. SIAS man-

ages tuple versions of a single data item as simply linked lists (chains), addressed by a virtual tuple ID (VID). Figure 1 illustrates the invalidation process in SI and SIAS. Transactions $T1$, $T2$, $T3$ update data item X in serial order. Thereafter, the relation contains three different tuple versions of data item X. The initial version X_0 of X is created by $T1$ and updated by $T2$. The *traditional approach (SI)* invalidates X_0 in-place by physically setting the invalidation timestamp and creating X_1. Analogously, $T3$ updates X_1 with the physical in-place invalidation of X_1. *SIAS* connects tuple versions using the VID where the newest tuple version is always known. Each tuple maintains a backward reference to its predecessor, which does not need to be updated in place. Hence, updating X_0 leads to the creation of X_1.

We report our work in progress on data placement and summarize key findings and the preliminary results of SIAS (published in a previous work). In this paper, we focus on novel aspects of *version handling*, (physical) *placement* and *collocation* on append-based database storage manager using flash memory as primary storage.

In Section II we present the related work. Section III provides a brief summary of the properties of flash technology. Section IV introduces the SIAS approach, aspects of *version handling*, (physical) *placement* and *collocation*. Section VI concludes the paper.

II. Related Work

SIAS organizes data item versions in simple chronologically ordered chains, which has been proposed by Chan et al. in [3] and explored by Petrov et al. in [4] and Bober et al. in [5] in combination with MVCC algorithms and special locking approaches. Petrov et al. [4], Bober et al. [5], Chan et al. [3] explore a log/append-based storage manager. The applicability of append-based database storage management approaches for novel asymmetric storage technologies has been partially addressed by Stoica et al. in [6] and Bernstein et al. in [7] using page-granularity, whereas SIAS employs tuple-granularity much like the approach proposed by Bober et al. in [5], which, however, invalidates tuples in-place. Given a page granularity the whole invalidated page is remapped and persisted at the head of the log, hence no write-overhead reduction. In tuple-granularity, multiple new tuple-versions can be packed on a new page and written together. Log storage approaches at file system level for hard disk drives have been proposed by Rosenblum in [8]. A performance comparison between different MVCC algorithms is presented by Carey et al. in [9]. Insights to the implementation details of SI in Oracle and PostgreSQL are offered by Majumdar in [10]. An alternative approach utilizing transaction-based tuple collocation has been proposed by Gottstein et al. in [11]. Similar chronological-chain version organization has been proposed in the context of update intensive analytics by Gottstein et al. in [12]. In such systems data-item versions are never deleted, instead they are propagated to other levels of the memory hierarchy such as HDDs or Flash SSDs and archived. Any logical modification operation is physically realized as an append. SIAS on the other hand provides mechanisms to couple version visibility to (logical and physical) space management. SIAS uses transactional time (all timestamps are based on a transactional counter) in contrast to timestamps that correlate to logical time (dimension). Stonebraker et al. realized the concept of TimeTravel in PostgreSQL [13]. A detailed analysis of append storage in multi-version databases on Flash is reported by Gottstein et al. in [2].

III. Flash Memories

The performance exhibited by Flash SSDs is significantly better than that of HDDs, yet Flash SSDs, are not merely a faster alternative to HDDs and just replacing them does not yield optimal performance. This section gives an extended discussion of their characteristics, as reported in [11].

(i) *asymmetric read/write performance* the read performance is significantly better than the write performance up to an order of magnitude. This is a result of the internal organization of the NAND memory, which comprises two types of structures: pages and blocks. A page (typically 4 KB) is a read and write unit. Pages are grouped into blocks of 32/128 pages (128/512KB). NAND memories support three operations: read, write, erase. Reads and writes are performed on a page-level, while erases are performed on a block level. A write is only possible to be performed on a clean (erased) block. Hence, before performing an overwrite, the whole block containing the page has to be erased, which is a time-consuming operation. Direct overwrites, as on traditional magnetic HDDs, are not possible. The respective flash memory raw latencies are: read-55s; write 500s; erase 900s. In addition, writes should be evenly spread across the whole volume. Hence, in-place updates as on HDDs are not possible, instead copy-and-write is applied.

(ii) *excellent random read throughput (I/O Operations per second – IOPS)* especially for small block sizes (as reported in [4]). Small random reads are up to hundred times faster than on an HDD. The good small block performance (4KB, 8KB) affects the present assumptions of generally larger database page sizes.

(iii) *low random write throughput*; small random writes are five to ten times slower than reads. Nonetheless, the random write throughput is an order of magnitude better than that of an HDD. Random writes are an issue not only in terms of performance but also yield long-term performance degradation due to Flash-internal fragmentation effects. Recent Flash device manufacteurs report faster random write than random read IOPS, but these figures can only be achieved by large on-device caches and do not consider sustained workload. As soon as their cache is filled, the performance of the Flash device is bound by the characteristic performance of the Flash memory.

(iv) *good sequential read/write transfer.* Sequential operations are also asymmetric. However, due to read ahead, write back and good caching the asymmetry is below 25%.

(v) *endurance issues and wear*; Flash memories are prone to wear. They only support a limited amount of erase cycles – avoiding in-place updates and reducing overwrites therefore aids longevity.

(vi) *suboptimal mixed load performance*: mixing reads/writes or random/sequential patterns leads to performance degradation.

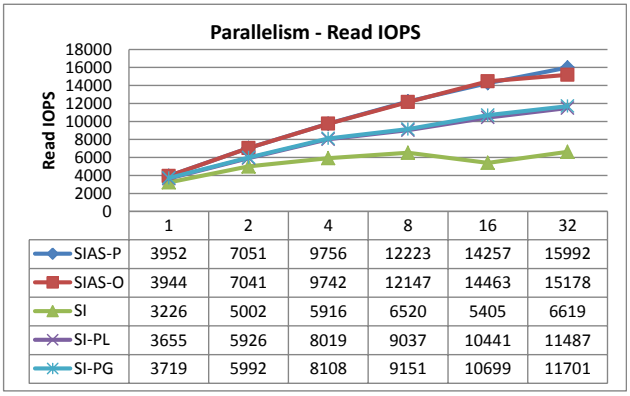
TABLE I. SIAS AND SI RESULTS ON INTEL X25-E SSD [14]

Queue Depth 1

Trace	read IOPS	write IOPS	read MB	write MB	time (sec)
SIAS-O(I)	4476	20	19713	89.96	563.675
SIAS-P(I)	4499	19	20666	89.96	587.873
SI (I)	3771	322	19901	1624	721.843
SIAS-O(II)	3947	13	11542	39.76	374.204
SIAS-P(II)	3953	13	11562	39.76	374.341
SI (II)	3656	432	11852	1395	414.869

Queue Depth 32

Trace	read I/O	write I/O	read MB	write MB	time (sec)
SIAS-O(I)	14500	66	19713	89.96	174.01
SIAS-P(I)	14642	63	20666	89.96	180.658
SI (I)	3360	264	19901	1624.9	805.193
SIAS-O(II)	15981	55	11542	39.76	92.44
SIAS-P(II)	15722	54	11562	39.76	94.128
SI (II)	11365	1338	11852	1395	133.478



	1	2	4	8	16	32
SIAS-P	3952	7051	9756	12223	14257	15992
SIAS-O	3944	7041	9742	12147	14463	15178
SI	3226	5002	5916	6520	5405	6619
SI-PL	3655	5926	8019	9037	10441	11487
SI-PG	3719	5992	8108	9151	10699	11701

Fig. 2. I/O Parallelism on Intel X25-E SSD - 60 Minute TPC-C

IV. SIAS - SNAPSHOT ISOLATION APPEND STORAGE

In this section we provide a summary of the SIAS approach [2]. SIAS manages versions as simply linked lists (chains) that are addressed by using a virtual tuple ID (VID), displayed in Figure 2. On creation of a new version it implicitly invalidates the old one resulting in an out-of-place write – implemented as a logical append – and avoiding the in-place update of the predecessor. The most recent version in the chain is known as the *entrypoint* of the chain. Without going into further details of the algorithm, the visibility is determined by accessing the entrypoint first and if it is not visible yet (long running transaction) the predecessor version is fetched using a pointer stored on the tuple version itself. In order to keep the entrypoint of each VID, SIAS employs a lightweight datastructure, where only an entry is created if the data item is comprised of more than one tuple version. SIAS is coupled to an append-based storage manager, appending in units of tuple versions and writing in granularities of pages. Only completely filled pages are appended in order to keep the packing dense, which is one of the reasons for the lower write amplification.

The example in Figure 2 shows the history of three transactions creating/updating data item X. The initial version is created by transaction T_1. Up to this point the traditional approach and SIAS create tuple version X_0. Transaction T_2 updates data item X. The traditional approach invalidates X_0 by stamping it with its own timestamp (in-place update) and creates a new version X_1 that points to X_0, analogously X_0 receives a pointer to X_1. SIAS creates the new version X_1 and stores it as the entrypoint. X_1 receives a pointer to X_0 and X_0 is left unchanged. X_1 is appended to the head of the log storage and written to the storage as soon as the page is completely filled or a arbitrary, pre-defined threshold is reached (WAL and recovery-mechanisms are left untouched). Subsequent updates proceed analogously.

Table I shows our test results with SIAS. Two traces containing all accessed and inserted tuples were recorded under PostgreSQL running TPC-C instrumented with different parameters. *Trace I* was instrumented using 5 warehouses with four hours runtime and *Trace II* using 200 warehouses and 90 minutes runtime. Both traces were fed into our database storage simulator, which generated SIAS-O/P and SI traces, containing read and written DB-pages to be used as input for the FIO benchmark, which executed them on an Intel X25-E SSD. SIAS-O is a simulation with and SIAS-P without caching of the SIAS data structures, where SI is the classic Snapshot Isolation using in-place updates on the invalidation. The conclusions of our results are:

(i) SI reads more than SIAS-O but less than SIAS-P

(ii) SI writes more gross-data than SIAS-O/P

(iii) SIAS-O/P reads with more IOPS than SI

(iv) SIAS needs less runtime than SI

(v) SIAS-O/P scales better than SI with higher parallelism.

We also conducted tests using SI and page-wise append, performing a remapping of all pages, which either appends pages local at each relation (SI-PL) or at a global append area (SI-PG) with the results displayed in Figure 2. Figure 3 illustrates the resulting write patterns using the blocktrace tool in Linux ($QD = 1$). They both achieve comparable performance in write throughput, nevertheless on subsequent read accesses the local approach has the advantage over the global approach - since the local approach makes better use of locality. This means that in the local approach pages of different relations are not interleaved as in the global approach. We found that in general both of the SI page append approaches outperform the original in-place SI by 15 to 76%, both themselfes are outperformed by SIAS-O/P by 6 to 36%. Our results empirically confirm our hypothesis that (a) appends are more suitable for Flash, (b) append granularity is crucial to performance and (c) appending in tuples and writing in pages is superior to remapping of pages. In the following sections we describe our approaches to merging of pages and physical tuple version placement as well as compression and indexing.

A. Write Amplification

One of our key benefits of the tuple based append log storage in SIAS is the significant reduction in write overhead. In our TPC-C benchmarks we observed a write reduction of up $52x$ compared to a traditional in-place update approach. The in-place update approach yields the same amount of write amplification as an append log storage manager that appends in the granularity of pages (page LbSM) – where the contents of each page are unknown.

Review the example in Figure 2, the traditional approach invalidates an old tuple version of data item X in place. This in-place update, even if it only updates a timestamp and a pointer, leads to the re-write of the page that contains the tuple version. In the example X_0 gets invalidated by transaction

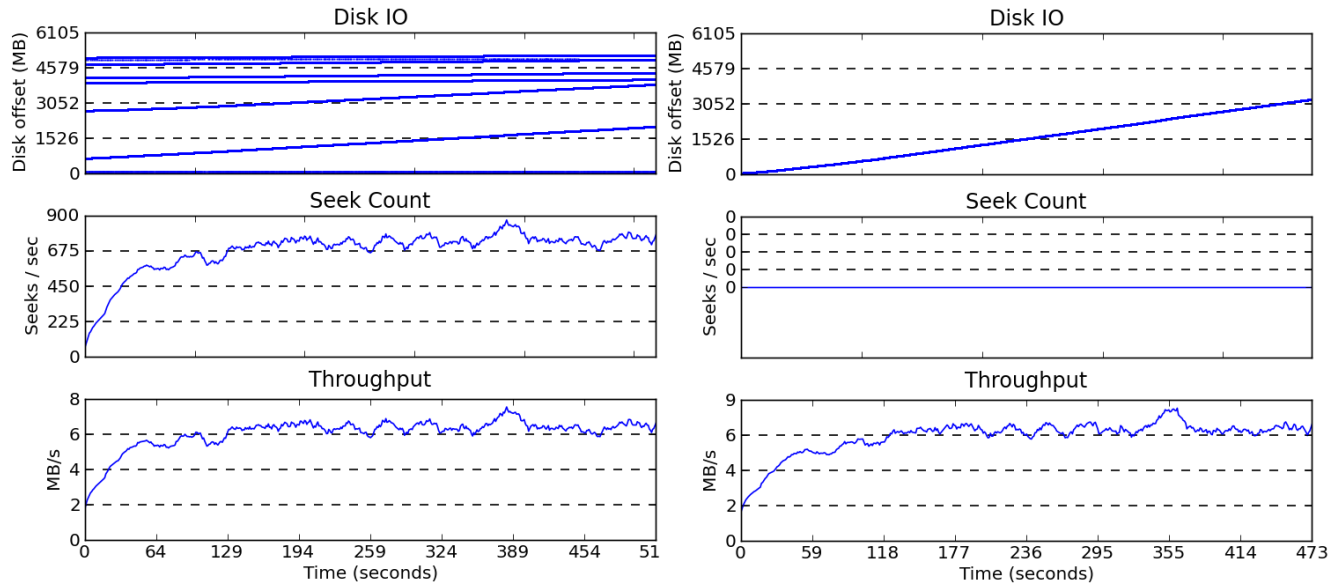


Fig. 3. Blocktrace: Left Local Append Regions (SI-PL) - Right Global Append Region (SI-PG)

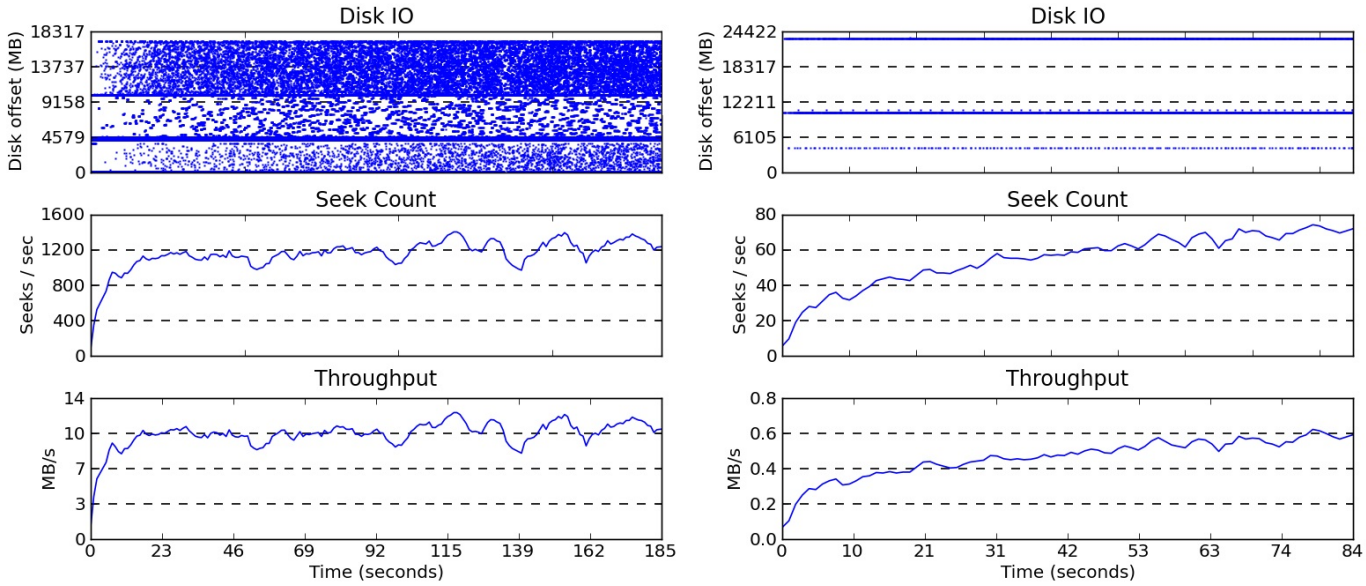


Fig. 4. Blocktrace: Write Overhead - Left In Place Update - Right SIAS

T_2, which leads to the re-write of the page that contains X_0. If the new version X_1 is stored in a different page, it also has to be written to stable storage. The update on X_0 only updates the visibility meta-information, which is necessary to determine the visible version of X, since older transactions are still able to read X_0. The traditional in-place update approach to multi-versioning, therefore, physically updates the predecessor version although the content did not change. Hence, a whole page may be re-written, which leads to a significant write amplification. One tuple version has to be inserted and in the worst case two pages have to be written. The page append storage manager transforms such in-place updates into appends, but still has to write the additional page. In SIAS this effect is alleviated by leaving the old version 'untouched'. Since the new tuple version is inserted into a new page, which is only written when it is filled (or an arbitrary threshold is reached) – this leads to the reported write reduction. The effect on the write pattern is displayed in a blocktrace diagram in Figure 4. The diagram shows the blocktrace of the default in-place update multi-versioning including its default in-place storage management and the SIAS algorithm. The workload was the resulting IO-Pattern of TPC-C trace configured with 10 clients, 200 warehouses and had a runtime of 90 minutes. The trace showed a 52 times write reduction when using SIAS (as reported in [2]). We found that the longer the trace, the higher the write reduction, which is a logical consequence since the amount of updates directly correlates with the reduction. It is also visible that the in-place approach needs more time

to complete the workload, while the append storage finishes earlier. The reason for the lower throughput in SIAS is the low amount of writes that have to be issued.

B. Merge

One key assumption of append based storage is that once data was appended it is never updated in-place. In a multi-version database old and updated versions inevitably become invisible, which leads to different tuple versions of the same data item, most likely located at different physical pages. Hence, pages *age* during runtime and contain visible and invisible tuple versions. In a production database running 24x7 it is realistic to assume that *net amount of visible tuples* on such pages is low and that an ample amount of outdated *dead* tuple versions is transferred, causing cache pollution. Once a certain threshold of dead tuples per page is reached it is beneficial to re-insert still visible tuples and mark the page as invalid. Dead tuples may be pruned or archived. Since a physical invalidation of the old page would lead to an in-place update, we suggest using a bitmap index providing a boolean value per page indicating its invalidation. The page address correlates to the position in the bitmap index, therefore, the size is reasonably small. A merge therefore includes the re-insertion of still visible tuples into a new page and the update of the bitmap index. On the re-insertion the placement of the tuples may be reconsidered (Sect. IV-C).

Space reclamation of invalidated pages is also known as garbage collection in most MVCC approaches. On flash memories, a physical erase can only be executed in erase unit granularities, hence it makes sense to apply reclamation in such granules and to make use of the *Trim* command. Pruning a single DB-page with the size smaller than an erase unit will most likely cause the FTL to create a remapping within the it's logical/physical block address table and postpones the physical erasure. This may result in unpredictable latency outliers due to fragmentation and postponed erasures [4]. Using the bitmap index, indicating deleted/merged pages (prunable), a consecutive sequence of pruned pages within an erase unit can be selected as a victim altogether. If the sequence still contains pages, which have not been merged yet, they can be merged before the reclamation.

SIAS uses data structures to guarantee the access to the most recent committed version X_v of a data item X, the *entrypoint*. If only the most recent committed version has to be re-inserted (i.e., no successor version exists), nothing but the SIAS data structure has to be updated. It is theoretically possible that the tuple version is still visible and invalidated. In this case a valid successor version to that tuple exists, which has to be re-inserted as well: Let P_m be the victim page, X_i an invalidated tuple version of data item X, where $X_i \in P_m$ and $X_v \in P_k$, $P_m \neq P_k$. X_v is the direct successor to X_i physically pointing to X_i. The merge of P_m leads to a re-insertion of X_i as X_i*, which leads to a re-insertion of X_v as X_v*, pointing to X_i*. The SIAS data structures are updated such that the most recent committed version of X know is X_i*. It is not necessary to merge P_k as well, since X_v simply becomes an orphan tuple version, which is not reachable by the SIAS data structures. Phantoms cannot occur since X_v* and X_v yield the same VID and version count. Nevertheless, it is most likely that X_i will become invisible during the merge since OLTP transactions are usually short and fast running. Further the structure is self contained on the tuples. On a crash it can be re-created by, e.g., a full sequential scan of the relation. The mapping of virtual ID, that identifies the data item, to tuple version id, which identifies the data item in a defined state in time can easily be created since each tuple version stores the VID. The existing methods of a write ahead log approach can be utilized to log changes in the SIAS datastructure.

C. Tuple Version Placement

In SIAS, each relation maintains a private append region and tuples are appended in the order they arrive at the append storage manager. Tuples of different relations are not stored into the same page and pages of different relations are not stored into the same relation regions. Appending tuple versions in the order they arrive may be suboptimal, since merged, updated and inserted tuples usually have different access frequencies. Collocation of tuples according to their access frequency can be benefitial since the net amount of actually used tuples per transferred page is higher [11]. Using temperature as a metric, often accessed tuples are hot and seldom accessed tuples are cold. The goal of tuple placement is to transfer as much hot tuples as possible with one I/O to reduce latency and to group cold tuples such that archiving and merging is efficiently backed. Visibility meta-information also contributes to access frequency, since tuples need to be checked for visibility. This creates yet another dimension upon which tuples can be related apart from the attribute values. Even if the content is not related the visibility of the tuples may be comparable.

Under the working set assumption and according to the 80/20 rule - both are the key drivers of data placement - (80% of all accesses refers to 20% of the data – as in OLTP enterprise workloads [15]) statistics can be used during an update to inherit access frequencies to the new tuple version.

In SIAS, the length of the chain describes the amount of updates to a data item (amount of tuple versions). Hence, a long chain is correlated to a frequently updated data item. A page containing frequently updated tuple versions will likely contain mostly invisible tuples after some runtime, hence simplifying the merge/reclamation process.

Version Meta Data Placement: Version metadata embodying a tuple's visibility/validity is stored on the tuple itself in existing MVCC implementations. An update creates a new version and version information of the predecessor has to be updated accordingly. SIAS benefits largely from the avoidance of the in-place invalidation. Further decoupling visibility information and raw data would be even more benefitial. Raw data becomes stale and redundancies caused by, e.g., tuples that share the same content but different visibility information are reduced or vanish completely. A structure that separately maintains all visibility information, enables accessing only needed data (payload) on Flash memory. This principle inherently deduplicates tuple data and creates a dictionary of tuple values. Visibility meta-information can be stored in a column-store oriented method, where visibility information and raw tuple data form a n:1 relation. This facilitates usage of compression and compactation techniques. A page containing

solely visibility meta-information can be used to pre-filter visible tuple versions, which subsequently can be fetched in parallel utilizing the inherent SSD parallelism, asynchronous I/O and prefetching.

Choosing the appropriate storage medium for this data is critical for performance, especially since new storage technologies change the traditional memory hierarchy augmenting it with new levels [16]. Non Volatile Memories such as Phase Change Memories seem to be a good match as they support: (i) in-place updates; (ii) fast random access (read and write); (iii) byte adressability; (iv) higher capacity than RAM. Byte addressability is important for small updates of, e.g., timestamps and to support differential updates. They still yield an inherent read/write asymmetry and are exposed to wear. A data structure within such a NVM can store pointers to raw data on flash. Our current work includes separation and placement of version information.

The SIAS data structure that stores a mapping of the virtual ID to the most recent tuple version of a single data item can be stored on such memories. In our current SIAS approach this lightweight datastructure is stored in main memory. In this way the properties of Flash memories are optimally addressed, since writes are only executed as appends and reads can be executed in parallel and smaller blocksizes. In SIAS tuple version only store stale version information, such as the creation timestamp and a pointer to the predecessor version (if the version is not the first of the data item). This also enables the usage of a multi versioned index structure that is capable of delivering the visibility decision by only accessing the index structure described in Section V.

D. Optimizations

A number of optimization techniques can be derived from observation that in append based storage a page is never updated, yet: compression, optimization for cache and scan efficiency, page layout transformation etc. Generally these facilitate analytical operations (large scans and selections) on OLTP systems supporting archival of older versions.

Compression. Most DBMS store tuples of a relation exclusively on pages allocated for that very relation. In a multi version environment, versions of tuples of that relation are stored on a page. Since all these have the same schema (record format) and differ on few attribute values at most, the traditional light-weight compression techniques (e.g., dictionary- and run-length encoding) can be applied.

Page-Layout and Read Optimizations. Since the content of a written page is immutable and only read operations can access the page, a number of optimizations can be considered. If large scans (e.g., log analysis) are frequent, cache efficiency becomes an issue, hence the respective page-layouts can be selected. Furthermore it is possible to use analytical-style page layout (e.g., PAX) for the version data and traditional slotted pages for the temporary or update intensive data such as indices. In [17] we analyse the effect of *sorted runs* in MV-DBMS' with ordered append log storage and multi-version index structures on Flash storage. It is benefitial to append in sorted runs rather than unsorted single pages and even more benefitial when it is implemented within the MV-DBMS, since the MV-DBMS is capable to use the inherent knowledge about the data. The parallelism of the Flash memories is leveraged by multiple write streams, created by the separation of append regions – each relation has its own (local) append region instead of one single (global) append region for alle relations.

V. Multi Version Index

Index structures are vital component of modern databases. Hence, their importance, especially as performance critical components, they are still a widely ignored aspect in MV-DBMS on asymmetric storage. Index structures are mostly not aware of versioned data and therefore, incapable to leverage their properties. Maintaining them on asymmetric storage becomes a critical issue.

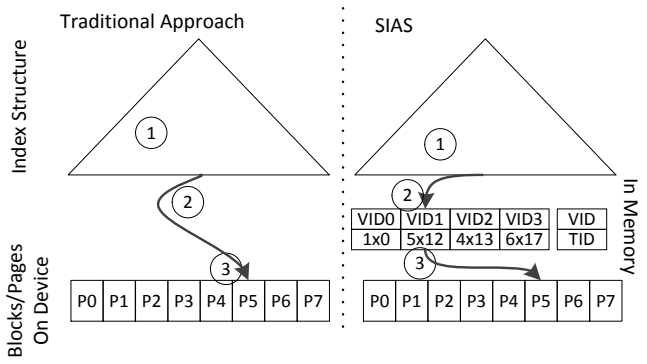


Fig. 5. Indexing: Traditional and SIAS

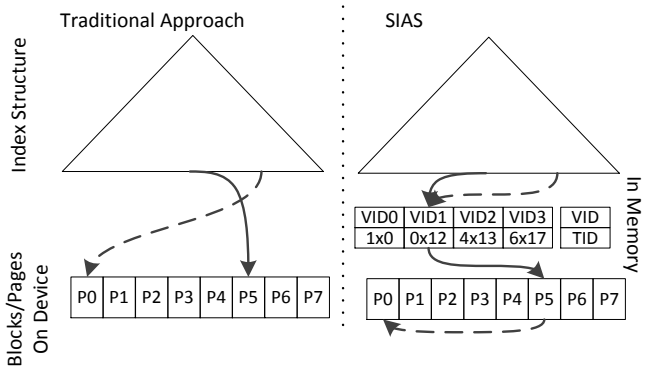


Fig. 6. Multi Version Indexing: Traditional and SIAS

Although data items exist in different tuple versions, the index addresses each version as a unique data item. This leaves the task of filtering visible versions to the rest of the MV-DBMS (e.g., executor, transaction manager). In the MV-DBMS updates of a data item lead to the out-of-place creation of a new tuple version. The index structure has to be updated (in-place) in order to index the correct tuple version that represents the data item.

The previous tuple version of a data can still be visible to some old running transactions, therefore, the index has to wait with the deletion of the pointer to the outdated version. If the index update is delayed there might be more than one tuple version of a single data item that matches the (indexed) search criteria. Hence, the index can return a data item in two different states (versions). Hence, the DBMS implementation

has to filter correct tuple versions of the data items after the access to the index. Since the visibility meta-data is stored on the tuple versions themselfes, this causes additional accesses to the I/O subsystem - even if *none* of the tuple versions is visible.

A. Index Structures in SIAS

SIAS identifies tuple versions of a single data item with a VID that is unique for all tuple versions belonging to that data item. Hence, the indexing problem can be fixed by storing the VID in the index, rather than the direct pointer to the tuple version. This gives us the benefit that indices do not have to be updated immediately when a new tuple version of a data item is created. The old entry points to the VID of the data item, which subsequently points to the most recent tuple version.

Figure 5 shows the index in the traditional approach and the SIAS algorithm. The traditional approach stores a pointer to the tuple version, treating it as a unique data item. Fetching a tuple version using such an index is comprised of 3 steps: first the index is searched using a search key. Second, assuming that a match has been found a pointer is followed. Third the tuple version is fetched and has to be checked against the visibilty criteria.

SIAS stores a pointer to the VID of the data item, which is redirected to the most recent tuple version of the data item. Fetching a tuple version is also comprised of three steps including one indirection. First, as in the traditional approach, the index structure is search using a search key. Second, assuming that a match is found, the SIAS datastructure is accessed and the pointer is followed to the most recent version. Third the tuple version is fetched and the SIAS algorithm determines the visibility.

In Figure 6, we assume that a data item exists in two tuple versions, which are both still visible. The first version of the data item is located on page $P0$ and the successor version is located on page $P5$. This case is most likely since under an LbSM approach new versions tend to be located at a position further in the log storage. In the traditional approach the index has two entries pointing to different positions on the disk. In SIAS both pointers will point to $VID1$, which stores the pointer to only the most recent version. In SIAS the index is capable of delaying updates, if now the version stored in $P0$ becomes invisible, the backwards pointer on the tuple version stored in $P5$ won't be followed and the version stored in $P5$ becomes the *stable* version of the data item. This means that the stable version is the tuple version of a data item that is committed and no running transaction is capable to read a previous version. In the traditional approach there is a tradeoff to pay, the visibility can be determined by accessing the version individually, which means that theoretically the deletion of the index entry for the old version can also be delayed but the cost of accessing the I/O storage always has to be payed.

1) Improvements on the Multi Version Index: Our current research is on the improvement of the index structure in order to be capable to answer all visibility related checks by only accessing the index structure. Hence, avoiding the access to a tuple alltogether. We have introduced an improvement that is capable of answering most of the visibility related checks by accessing the index only in [17].

VI. Conclusion and Future Work

We propose the combination of multi-version databases and append-based storage as most beneficial to exploit the distinguishing characteristics new storage technologies (Flash, NVM). When integrated they help: (i) utilise the excellent read performance low read latencies of such technologies for validity and visibility checks as well as due to the fact that readers are never blocked by writes; (ii) in addition, several types of read optimisation can be performed on LbSM level; (iii) the out-of-place update semantics resulting from the fact that upon a tuple update a new physical version is produced can be successfully utilised to reduce the expensive random writes resulting from in-place updates; (iv) existing algorithms have been revised to enable these changes.

We have prototypically implemented SIAS in PostgreSQL and validated the reported simulation results. The highest performance benefit can be achieved by the integration of the append storage principle directly into a multi-version DBMS, reducing the update granularity to a tuple-version, implementing all writes out-of-place as appends, and coupling space management to version visibility. In contrast page remapping append storage manager does not fully benefit of the new storage technology. SIAS is a Flash-friendly approach to multi-version DBMS: (i) it sequentialises the typical DBMS write patterns, and (ii) reduces the net amount of pages written. The former has direct performance implications the latter has long-term longevity implications.

In addition, SIAS introduces new aspects to data placement making it an important research area. We especially identify version archiving, selection of hot/cold tuple versions, separation of version data and version meta-data, compression and indexing as relevant research areas.

In our next steps, we focus on optimizations such as compression of tuple versions to further reduce write overhead by 'compacting' appended pages, placement of correlated tuple versions to increase cache efficiency as a 'per page clustering' approach and an efficient indexing of multi-version data using visibility meta-data separation.

Acknowledgment

This work was supported by the DFG (Deutsche Forschungsgemeinschaft) project "Flashy-DB".

References

[1] R. Gottstein, I. Petrov, and A. Buchmann, "Aspects of Append-Based Database Storage Management on Flash Memories," in *DBKDA 2013, The Fifth International Conference on Advances in Databases, Knowledge, and Data Applications*, 2013, pp. 116–120.

[2] ——, "Append storage in multi-version databases on Flash," in *BNCOD 2013, British National Conference on Databases*. Springer Berlin Heidelberg, 2013, pp. 62–76.

[3] A. Chan, S. Fox, W.-T. K. Lin, A. Nori, and D. R. Ries, "The implementation of an integrated concurrency control and recovery scheme," in *19 ACM SIGMOD Conf. on the Management of Data, Orlando FL*, Jun. 1982.

[4] I. Petrov, R. Gottstein, T. Ivanov, D. Bausch, and A. P. Buchmann, "Page size selection for OLTP databases on SSD storage," *JIDM*, vol. 2, no. 1, pp. 11–18, 2011.

[5] P. Bober and M. Carey, "On mixing queries and transactions via multiversion locking," in *Proc. IEEE CS Intl. Conf. No. 8 on Data Engineering, Tempe, AZ*, feb 1992.

[6] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki, "Evaluating and repairing write performance on flash devices," in *Proc. DaMoN 2009*, P. A. Boncz and K. A. Ross, Eds., 2009, pp. 9–14.

[7] P. A. Bernstein, C. W. Reid, and S. Das, "Hyder - A transactional record manager for shared flash," in *CIDR*, 2011, pp. 9–20.

[8] M. Rosenblum, "The design and implementation of a log-structured file system," U.C., Berkeley, Report UCB/CSD 92/696, Ph.D thesis, Jun. 1992.

[9] M. J. Carey and W. A. Muhanna, "The performance of multiversion concurrency control algorithms," *ACM Trans. on Computer Sys.*, vol. 4, no. 4, p. 338, Nov. 1986.

[10] D. Majumdar, "A quick survey of multiversion concurrency algorithms," 2006. [Online]. Available: "http://forge.objectweb.org/docman/view.php/237/132/mvcc-survey.pdf"

[11] R. Gottstein, I. Petrov, and A. Buchmann, "SI-CV: Snapshot isolation with co-located versions," in *in Proc. TPC-TC*, ser. LNCS. Springer Verlag, 2012, vol. 7144, pp. 123–136.

[12] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. C. H. Plattner, P. Dubey, and A. Zeier, "Fast updates on read-optimized databases using multi-core CPUs," in *Proceedings of the VLDB Endowment*, vol. 5, no. 1, sep 2011.

[13] M. Stonebraker, L. A. Rowe, and M. Hirohama, "The implementation of postgres," *IEEE Trans. on Knowledge and Data Eng.*, vol. 2, no. 1, p. 125, Mar. 1990.

[14] R. Gottstein, I. Petrov and A. Buchmann, "SIAS: Chaining Snapshot Isolation and Append Storage," submitted.

[15] S. T. Leutenegger and D. Dias, "A modeling study of the TPC-C benchmark," in *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1993, p. 22.

[16] I. Petrov, D. Bausch, R. Gottstein, and A. Buchmann, "Data-intensive systems on evolving memory hierarchies," in *Proc. of Workshop Entwicklung energiebewusster Software (EEbS 2012), 42. GI Jahrestagung*, 2012.

[17] R. Gottstein, I. Petrov, and A. Buchmann, "Read Optimisations for Append Storage on Flash," in *IDEAS 13, 17th International Database Engineering and Applications Symposium*, 2013.