# SIAS-Chains: Snapshot Isolation Append Storage Chains


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

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

Sergey Hardock
Databases and Distributed Systems Group
TU-Darmstadt
Darmstadt, Germany
hardock@dvs.tu-darmstadt.de

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



## ABSTRACT

Asymmetric read/write storage technologies such as Flash are becoming a dominant trend in modern database systems. They introduce hardware characteristics and properties which are fundamentally different from those of traditional storage technologies such as HDDs.

Multi-Versioning Database Management Systems (MV-DBMSs) and Log-based Storage Managers (LbSMs) are concepts that can effectively address the properties of these storage technologies but are designed for the characteristics of legacy hardware. A critical component of MV-DBMSs is the invalidation model: commonly, transactional timestamps are assigned to the old and the new version, resulting in two independent (physical) update operations. Those entail multiple random writes as well as in-place updates, sub-optimal for new storage technologies both in terms of performance and endurance. Traditional page-append LbSM approaches alleviate random writes and immediate in-place updates, hence reducing the negative impact of Flash read/write asymmetry. Nevertheless, they entail significant mapping overhead, leading to write amplification.

In this work we present an approach called Snapshot Isolation Append Storage Chains (SIAS-Chains) that employs a combination of multi-versioning, append storage management in tuple granularity and novel singly-linked (chain-like) version organization.

SIAS-Chains features: simplified buffer management, multi-version indexing and introduces read/write optimizations to data placement on modern storage media. SIAS-Chains algorithmically avoids small in-place updates, caused by in-place invalidation and converts them into appends. Every modification operation is executed as an append and recently inserted tuple versions are co-located.


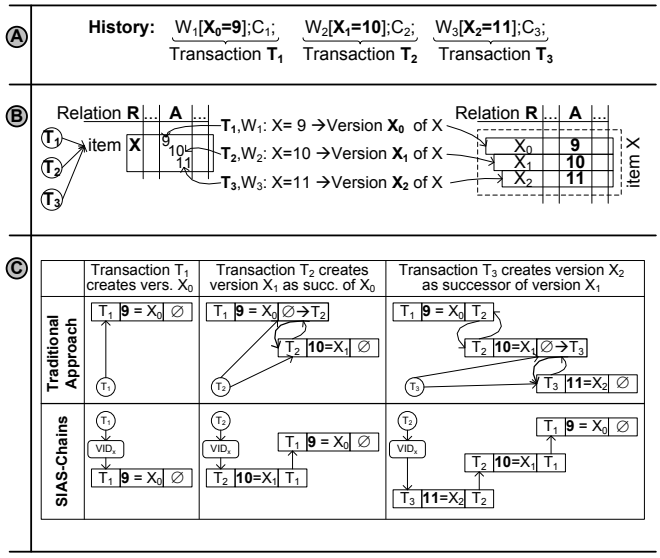


**Figure 1: SIAS-Chains and Traditional (SI) Invalidation: (A) Transactions $T_1$, $T_2$ and $T_3$ update data item X; (B) Data item X now comprises 3 versions; (C) Version organization and invalidation scheme under the traditional (SI) approach as a doubly-linked list, causing write overhead due to version invalidation and SIAS-Chains as a singly-linked list, where new versions are appended thus optimizing write I/O.**

SIAS-Chains is implemented in PostgreSQL and evaluated on modern Flash SSDs with standard update-intensive workload. The performance evaluation under PostgreSQL shows: (i) higher transactional throughput - up to 30%; (ii) significantly lower response times - up to 7 times lower; (iii) significant write reduction - up to 97% reduction; (iv) reduced space consumption and (v) higher tolerable workload.

# 1. INTRODUCTION

Database approaches and techniques based on multi-versioning have been around for a while and enjoy wide system support (both commercial and open-source). Under such approaches multiple versions of each data item may physically co-exist, whereas every transaction operates against a snapshot of the database comprising all versions it is allowed to see for consistent execution. Read operations simply operate on the latest committed version visible to them and are therefore never blocked, yielding good read performance. An update operation produces a new version of the updated data item and invalidates the predecessor version. There are several factors, that contribute to the proliferation of multi-version techniques: 1. Their characteristics are in-line with enterprise workloads [24]. 2. Multi-version approaches inherently account for the properties of asymmetric read/write storage technologies (such as Flash or Non-Volatile Memories). Common properties of these technologies that distinguish them from traditional storage technologies (RAM, spinning disks) are: (i) read/write asymmetry (reads are much faster than writes); (ii) high I/O parallelism; (iii) low (random) write performance; (iv) endurance issues and wear.

Snapshot Isolation (SI) is introduced in Berenson, Bernstein et al. [6] and defines the general concurrency control algorithm making no claims about the physical version organization. Every tuple version is augmented with two timestamps (creation- and invalidation-timestamp) that are part of the physical version record. An updating transaction modifying a data item sets its transactional timestamp as the invalidation timestamp on the old version as well as the creation timestamp on the new version (Figure 1). Hence old versions of a data item are physically invalidated (in-place) as new versions are created and different versions of a data item form a logical order based on timestamps (creation- and invalidation-timestamp). The version invalidation implies two page updates, which in the general case are performed as in-place updates and result in random writes. Although several design alternatives exist [25], many systems implement it precisely as described above, which is suboptimal for asymmetric storage technologies.

Figure 1 illustrates the invalidation process in SI and SIAS-Chains: Following the history (Figure 1.A ), three Transactions ($T_1, T_2, T_3$) update data item $X$ in serial order resulting in a relation that contains three different tuple versions of data item $X$. $T_1$ creates the initial version $X_0$ of $X$. $T_2$ issues the first update (Figure 1.B). Under the traditional approach (SI), $X_0$ is invalidated in-place by setting its invalidation timestamp, subsequently $X_1$ is created. The update issued by $T_3$ proceeds analogously and $X_1$ is invalidated in-place while $X_2$ is created as a new tuple version (Figure 1.C).

In contrast, SIAS-Chains creates a new version, containing a backwards reference to its predecessor. Hence updating $X$, results in the creation of $X_1$, which implicitly invalidates $X_0$ avoiding the need of physical in-place modification (Figure 1.C). All versions of a given data item $X$ contain the same unique virtual tuple ID ($VID_x$) and form a simply-linked list (version chain). The newest version is always known and is called an *entrypoint* to the chain (data item). This will be explained in more detail in Section 4. The SIAS-Chains LbSM appends just the newly created versions $X_0$, $X_1$ and $X_2$ to a reserved database page. Once a given threshold is reached the page is appended to stable storage, resulting in significantly fewer write I/Os. In this paper we exploit the important fact that versions are separate entities, which can be individually moved, (physically) grouped, read and checked for visibility. The goal is to facilitate out-of-place updates, address read/write asymmetry and improve endurance. The contributions of the present paper are as follows:

(i) We propose the new Snapshot Isolation Append Storage - Chains (SIAS-Chains) algorithm, which treats the sequence of tuple versions as a singly-linked list (Figure 1) combining it with an LbSM. SIAS-Chains LbSM operates on fine-granular append units – tuple versions. It alleviates random writes and immediate in-place updates, hence reducing the impact of Flash read/write asymmetry. In contrast, traditional LbSMs introduce mapping and space management overhead as well as I/O granularity issues, leading to write amplification.

(ii) The *version invalidation scheme* is a vital part of the MV-DBMS: tuple versions of a data item are marked with timestamps to facilitate versioning. Each modification of a data item leads to a creation of a new tuple version and the invalidation of the old tuple version. Under Snapshot Isolation (SI) the old version is marked with an invalidation timestamp, leading to a physical (in-place) update (Figure 1). The SIAS-Chains approach introduces a new invalidation scheme leaving the old tuple version unchanged. SIAS-Chains no longer follows the paradigm of addressing each tuple version individually: multiple versions of a tuple are addressed as a unit. In this context, we also propose new criteria for version visibility.

(iii) SIAS-Chains is implemented in PostgreSQL, which required a major database redesign. The experimental evaluation under TPC-C-style workload and different Flash and HDD storage settings indicates: (a) higher transactional throughput (2x); (b) significantly lower response times; (c) significant write reduction (97% less issued write I/Os); and (d) reduced space consumption.

The rest of the paper is organized as follows. In Section 2 we present related approaches. A short introduction of MV-DBMSs is given in Section 3. We describe the SIAS-Chains algorithm in Section 4, providing comparison and delimitation to SI and illustrating its functionality with several examples. Section 5 presents the experimental evaluation results and we conclude with a summary in Section 7.

# 2. RELATED WORK

The concept of organizing data item versions in simple chronologically ordered chains has been proposed in [12] and explored in [27, 9] in combination with MVCC algorithms and special locking approaches. [27, 9, 12] explore a log/append-based storage manager. The authors also investigate approaches to garbage collection. The applicability of append-based database storage management approaches for novel asymmetric storage technologies such as Flash devices has been partially addressed in [33, 7, 32, 8, 30]. A common feature of these approaches is that they use page-granularity, whereas SIAS-Chains employs tuple-granularity much like the approach proposed in [9], which however invalidates tuples in-place. Given a page granularity the invalidated page still needs to be remapped in a holistic, global mapping and persisted, hence no write-overhead reduction. Given tuple-granularity, multiple new tuple-versions can be packed on a new page and written together at once. In the area of operating systems log storage approaches at file system level for hard disk drives have been proposed in [29].

Snapshot Isolation (SI) is presented in [6]. An overview of its implementation in PostgreSQL is given in [31]. Standard SI does not provide serializability [6]. Recently, serializable SI was proposed in [10], based on read/write dependency testing in serialization graphs. The PostgreSQL implementation of serializable SI is described in [28].

SI and MVCC enjoy broad system support. It has been implemented in many commercial and open-source systems: Oracle, IBM DB2, PostgreSQL, Microsoft SQL Server 2005, Oracle Berkeley DB, Ingres. In some systems SI is a separate isolation level, in others it is used to handle serializable isolation.

A performance comparison between different MVCC algorithms is presented in [11, 25, 36]. Approaches focusing on in-memory databases are studied in [23], [7]. They employ variants of versioning which are orthogonal to the work presented here. The Hekaton in-memory database (IM-DBMS) describes a different approach to serializability in [14]. Another approach to IM-DBMSs is proposed in [35]. SAP HANA also relies on multi-versioning and aims at real time business intelligence [15]. An alternative approach utilizing transaction-based tuple co-location has been proposed in [18].

Similar chronological-chain version organization has been proposed in the context of update intensive analytics [24]. In such systems data-item versions are archived and never deleted. Cleanup and version garbage collection are never performed due to the analytical nature of the system. SIAS-Chains provides mechanisms to couple version visibility to (logical and physical) space management and uses SIAS-Chains transactional time, in distinction to timestamps that correlate to logical time (dimension). Such features were first realized by Stonebraker et al. in PostgreSQL as the concept of TimeTravel [34]. To the best of our knowledge none of the used concepts e.g. append-based storage and MVCC have been explored for SI, which marks the contribution of this paper.

An overview of Flash storage properties is given in [13], design and performance tradeoffs are discussed in [5]. [26] gives an overview of MV-DBMSs on modern storage. Append storage (LbSM) in MV-DBMSs on Flash is discussed in [19], [17] and [20]. [21] contains read optimizations for LbSM in multi-version databases on Flash. [16] algorithmically integrates LbSM in tuple granularity into the MV-DBMS. It is demonstrated in [16] (video available online [3]).

SIAS-Chains introduces a new paradigm to version management, improves on the version invalidation scheme and adapts the indexing scheme. SIAS-Chains co-locates recently inserted tuple versions, an alternative approach to tuple version placement utilizing transaction-based tuple co-location has been proposed in [18]. [22] demonstrates an on-line Flash simulator, delivering direct access to Flash chips.

# 3. MULTI VERSION DBMS

MV-DBMSs are in line with the properties of the new storage technologies: (i) reads are never blocked by writes - they benefit from fast random reads and I/O parallelism of Flash; (ii) updates can conceptually be made out-of-place - circumventing the in-place update issue.

In MV-DBMSs tuples are associated with non-empty sets of tuple versions rather than a singular tuple representation as it is in traditional update-in-place DBMSs. Each modification of a data item creates a new tuple version of that item and the old version is invalidated. Whenever a transaction accesses a data item, the appropriate (visible) tuple version is returned by the DBMS. Visibility information is stored on each tuple version to facilitate finding the appropriate tuple version of the data item (see Section 4.1.1). In SI the visibility information is comprised of transactional timestamps: a creation timestamp inherited from the transaction that inserted the tuple version and an invalidation timestamp inherited from the transaction that invalidated it.

The invalidation model influences how the underlying storage is managed. When a tuple version is invalidated in-place, e.g. by marking it with a timestamp, the conceptual advantage of the out-of-place update is lost. The invalidation results in a small in-place update of the visibility information that is stored on the tuple version itself (a detailed example is discussed in Section 4.1) or in at least 2x write-amplification due to copy-on-write if a LbSM is used. Existing invalidation mechanisms rely on in-place invalidation of tuple versions. To update the visibility information of a single tuple version, the whole page (block) where the tuple version is stored has to be updated - which is suboptimal for Flash.

# 4. SIAS-CHAINS

SIAS-Chains no longer follows the paradigm of addressing each tuple version individually: tuple versions belonging to a certain data item are addressed as a whole since all of them receive the same unique identifier - *virtual ID (VID)*. The successor of a tuple version stores a reference to the predecessor version's physical location. Hence a new-to-old singly-linked version chain is constructed. In-place updates are thus conceptually avoided. The most recent (newest) tuple version is always known and called the *entrypoint* of the data item. Detailed examples are available in Sections 4.1 and 4.1.2. In Section 4.1.2 we discuss the requirements of and deliver our solution for a data structure that is capable of efficiently storing necessary information for the SIAS-Chains algorithm, both in terms of performance and space efficiency.

## 4.1 SIAS-Chains - Algorithm

Figure 1 depicts an example of the traditional and the SIAS-Chains invalidation scheme. In this example data item $X$ of relation $R$ is initially created in tuple version $X_0$ by transaction $T_1$. $T_2$ updates $X_0$ and creates the new tuple version $X_1$. Finally it is updated by $T_3$ which creates $X_2$. The relation therefore comprises three different tuple versions of $X$. In the traditional approach $X_0$ and $X_1$ are invalidated in place: $T_2$ fetches the page of $X_0$ and updates it in-place by setting the invalidation timestamp accordingly; this page will be later written back. Analogously $T_3$ updates $X_1$ (see on-tuple information - Section 4.1.1).

In SIAS-Chains the creation of a successor implicitly invalidates the previous version: $X$ receives a unique identifier, equal on all its tuple versions ($VID_x$). The mapping structure ($VID_{map}$, Section 4.1.2) always points to the *entrypoint* of $X$, which contains information about the previous tuple version (if one exists). Each tuple version $X_v$ is augmented with visibility information:

### 4.1.1 On-Tuple Information

(i) The creation timestamp $X_{v.create}$ which is denoted by the inserting transaction's ID; (ii) A unique $VID$ which is equal among all tuple versions of the data item it represents; (iii) A pointer $*ptr$ which stores the physical reference to an existing predecessor version, or $NULL$ if no such version exists; (iv) Tuple attributes such as type, format and attribute values. There is explicitly no invalidation information stored on each tuple version. The chained structure of the data item's tuple versions *code* this information along the version chain. The creation timestamp of the following version equals the invalidation timestamp of its successor version, as it is in the original snapshot isolation algorithm. The tuple versions of a single data item form a chronologically sorted chain. In Figure 1 the *entrypoint* is $X_2$, after $T_3$ has committed. $X_2$ is *chained* to the predecessor tuple version $X_1$.

### 4.1.2 Data Structures

SIAS employs the $VID_{map}$ data structure that stores a mapping of each VID to the *entrypoint* of the corresponding data item. There exists exactly one $VID_{map}$ for each relation which is used for all access paths. Each new data item receives a unique VID. The VIDs are increasing positive numbers, which are unique for all tuple versions of a single data item. Although a simple array is capable of holding the mapping functionality, the efficiency of the $VID_{map}$ is important. The requirements for the $VID_{map}$ involve

the support of fast exact match lookups, a low memory footprint, fast updates and the support for short time latches. The latches are necessary for updates, when a new tuple version is created and becomes the entrypoint of the data item's version chain. On an insert new values are appended. The mapping table might easily become a performance bottleneck for large database sizes, hence it should scale with data volume.

The search key in the $VID_{map}$ is the data item's $VID$ and the output is the corresponding $TID$ (Tuple version ID) of the entrypoint. Pre-loading and bulk-loading can be supported, e.g. new $VID$s can be generated in a page-wise manner. Assumptions about the page size and length of the $TID$ are implementation specific. The concept is applicable to different implementation choices. Our prototype uses the following configuration:

i) $TID$s are stored in pages of 8KB. ii) One $TID$ (in PostgreSQL) has the size of 6 Bytes and comprises the DB BlockID (32bit) and an offset to the tuple version (16 bit). iii) The maximum amount of $TID$s that fit into a page is 1365, exclusive header. iv) We store a maximum of 1024 $TID$s per page, a 10bit offset per TID is used. v) The record format is one $TID$ (of the latest version) per $VID$.

### 4.1.3 Hashtable

On large DB sizes the mapping may not fit completely into main memory and therefore parts of it need to be swapped to disk. The $VID_{map}$ is augmented with page abstraction (buckets). The bucket size equals the database page size. Since $VID$s of a relation are sequentially assigned, the buckets get filled sequentially. The position of each $TID$ within a bucket can exactly be calculated. The bucket number is determined by a *DIFF* operation on the VID and the capacity of the bucket: $BucketNr = \lfloor \frac{VID}{1024} \rfloor$.

The position within the bucket can be calculated using the modulus of 1024: $TID_{POS} = VID \bmod 1024$

There are no overflow buckets, since each $VID$ is in ascending order and has exactly one corresponding $TID$ (record format). Each update of a $TID$ indicates a new tuple version and substitutes the old $TID'$. The capacity of each bucket correlates with the hash function. A new bucket is allocated after each 1024 consecutive VIDs. Thus queries on VID ranges are also facilitated.

Concurrency: On insertions of new data items (assignment of a new $VID$) as well as on updates the corresponding storage slot within a hash bucket is currently latched. Latching can be avoided by using atomic instructions (e.g. CAS) as it is not algorithmically needed by the proposed hash-table variant.

The access cost $C_R$ for fetching a value is: ***O(1) + CPU***. The update cost $C_W$ in the hash table is the calculation of the position, setting/unsetting the latch and writing the new value: $2 * C_R$.The buffer has to be accessed and dirtied, hence the two-fold notation of $C_R$. The CPU denotes the cost to calculate the TID's position.

## 4.2 Access Methods and Operations

### 4.2.1 Scan

On a scan in SIAS-Chains the $VID_{map}$ is accessed first to determine visible tuple versions. *Note: This access path is parallelizable and therefore complements the parallelism of the Flash storage.*

For each $VID$ the visible tuple version is determined, rather than reading the complete set of tuple versions contained in the relation and subsequently checking each for visibility. In line 18 the visibility check is executed: the requirement is that the tuple version $X_v$ was committed before the checking transaction $tx$ started. Beginning with the *entrypoint* in the chain of each data item, the algorithm returns the first tuple version found that satisfies the visibility criteria. The tuple version stores its creation timestamp $X_{v.create}$ which corresponds to the inserting transaction's timestamp. If this timestamp is smaller or equal ($X_{v.create} \leq tx_{id}$, Algorithm 1, line 18) and the corresponding transaction was not concurrently running ($X_{v.create} \notin tx_{concurrent}$), then the tuple version was committed before the start of the accessing transaction - hence it is "visible. Structure $tx_{concurrent}$ records concurrently running transactions.

**Algorithm 1** SIAS-Chains Scan over $VID_{map}$

```
1:  procedure SCAN(Transaction tx)
2:      For each VID v ∈ VID_map {
3:      TupleVersion e=VID_map[v_entrypoint]          ▷ Entrypoint
4:      if isVisible(e, tx) then
5:          return e;
6:      else
7:          while(e_*ptr != null){
8:          p = fetch(e_*ptr)                         ▷ Predecessor
9:          if isVisible(p, tx) then
10:             return p
11:         else
12:             e = p
13:         end if
14:         }
15:     end if
16:     v=v.next();}                                  ▷ Next Data Item
17: end procedure
18: procedure ISVISIBLE(TupleVersion X_v, Transaction
    tx){
19:     return (X_v.create ≤ tx_id)&& (X_v.create ∉ tx_concurrent)
20:     }
21: end procedure
```

The traditional implementation, which was developed for HDDs to enable sequential I/O, first reads the whole relation (all tuple versions) and subsequently each tuple version is checked individually. Since SSDs enable fast random reads, the traditional scan is inefficient, since each tuple version has to be checked. SIAS-Chains scans the $VID_{map}$ first and enables more selective I/O as shown in the evaluation (Section 5.1). Nevertheless, on the traditional scan the same visibility criteria as in Algorithm 1 are applied. Since such a relation scan fetches all the tuple versions, each of them has to be checked for visibility individually it becomes a visible *candidate*. The same base algorithm is executed: the entrypoint of the data item is fetched and the visible version (if it exists) is determined as in Algorithm 1 - this version is compared with the *candidate* tuple version. If it matches, the check returns $true$. Obviously this method is not as efficient as the scan using only the entrypoints, since all of the potential versions of a data item need to be checked wich incurs additional memory consumption and CPU costs.

### 4.2.2 Insert - Update - Delete

During the *insertion* of a new data item $X$ its first tuple version $X_0$ is created. A new $VID$ is assigned: $VID_{map}$ stores the physical pointer to $X_0$. The on-tuple information is set: $X_{0.create}$ is set to the inserting transaction's ID (timestamp); $X_{0.*ptr}$ is set to $NULL$ and $X_{VID}$ is set accordingly.

An *update* proceeds similarly to an insertion. On an update of data item $X$ a new tuple version $X_n$ is created. All on-tuple information is analogously set as in an insert, except for the $*ptr$ variable, which is set as the physical pointer to the previous tuple version $X_{n-1}$. The $VID$ is inherited from the data item (equal among all versions of $X$). The entry within the $VID_{map}$ is set to the phys-

**Algorithm 2** SIAS-Chains Insert

```
1: procedure INSERT(DataItem X, Transaction tx)
2:     tx.lockX = REQUESTXLOCK(X)                    ▷ Lock for insert
3:     X_0 = new version(X)
4:     X_{0.create} = tx_id; X_{n.pred.create} = null;
5:     X_{0.*ptr} = null;                            ▷ No older version
6:     X_{0.VID} = getNewUniqueVID();                ▷ get new unique VID
7:     VID_map[X_{0.VID}] = X_{0.self};
8:     ON (tx.commit() or tx.rollback()): UpdateLog;
9:     Release aquired Locks; WakeUp waiting transactions;
10: end procedure
```

ical location of the new tuple version $X_n$, which now becomes the *entrypoint*. Only entrypoints are allowed to be updated and concurrent updates are avoided. The SIAS-Chains algorithm implements the *first-updater-wins* rule: An update in progress creates a new entrypoint of the data item which is not visible for concurrently running transactions - this "locks" the data item for updates of other transactions. Our implementation in PostgreSQL uses transaction locks, which deliver the desired functionality (Algorithm 3 line 7).

**Algorithm 3** SIAS-Chains Update

```
1: procedure UPDATE(TupleVersion X_u, Transaction tx)
2:     TupleVersion X_e = null;
3:     X_e= VID_map[X_{u.entrypoint}];               ▷ Entrypoint of X
4:     if (NOT((X_e==X_u) && (isVisible(X_e, tx)))) then
5:         tx.ROLLBACK( );
6:     end if
7:     tx.lockX = REQUESTXLOCK(X)                    ▷ Lock for update
8:     if (tx.lockX == GRANTED) then
9:         X_n = new version(X)
10:        X_{n.create} = tx_id; X_{n.pred.create} = X_e.create;
11:        X_{n.*ptr} = X_{e.self};                  ▷ Pointer to old entrypoint
12:        X_{n.VID} = X_{e.VID};
13:        VID_map[X_{n.VID}] = X_{n.self};
14:    else
15:        TX.WAIT(tx.lockX) if GRANTED goto[1]
16:    end if
17:    ON (Ti.commit() or Ti.rollback()): UpdateLog;
18:    Release aquired Locks; WakeUp waiting transactions;
19: end procedure
```

A *deletion* of a data item leads to the insertion of a special *tombstone* tuple version, necessary as long as there are running transactions, capable of "viewing" older tuple versions. For example: a transaction that started before the data item has been deleted by another transaction still has to access the last committed state of the data item (most recent tuple version) before the deletion.

## 4.3 SIAS-Chains - Indexing

When assuming a $B^+$ tree index on a relation R, the index records are traditionally comprised of a $< key, TID >$ pair. Since SIAS-Chains identifies all versions of a data item by using a $VID$, the index record is comprised of a $< key, VID >$ pair. Analogously to the $B^+$ tree, other index structures, e.g. Hash based index structures, can equally be adapted to the SIAS-Chains algorithm. For each relation there exists exactly one $VID_{map}$. SIAS-Chains uses the very same $VID_{map}$ for all access paths. Figure 2 depicts the indexing in SIAS-Chains. A $B^+$ tree index is created on attribute $A$ of relation $R$. The $VID_{map}$ serves to determine the *entrypoint* of the data item.

*Example 1: The value of an indexed attribute changes*: Transaction $T_1$ creates the initial version $X_0$ of data item $X$ with the attribute value 9. $T_2$ updates $X$, changed the value to 10 and creates $X_1$. The $B^+$ tree stores the $< key, VID >$ pairs, while the $VID_{map}$ structure points to the latest tuple version $X_1$, resident on $P5$. A reading transaction $T_4$ executes a lookup of all values in $R$ that match either 9 or 10. Using the $VID_{map}$ as a mediator, the $entrypoint$ is fetched from page $P5$. Note: if a transaction is *old enough* to not see $X_1$ but *young enough* to see $X_0$, the reference pointer on $X_1$ is used to fetch the previous version.

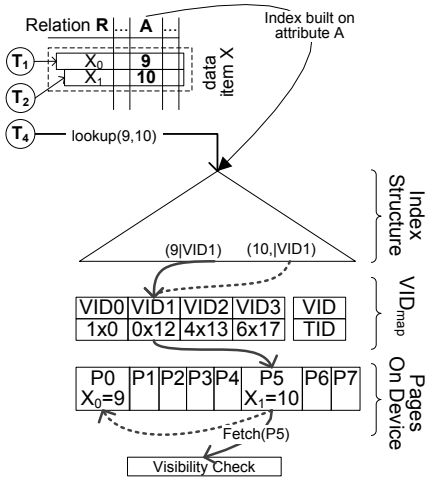


**Figure 2: SIAS-Chains Indexing: Index Key Update**

*Example 2: Update on a non-indexed attribute*: This example describes an update that does not involve a change of the index key value. Data item $X$ is inserted as in *Example 1* with the difference that the attribute value has not changed. For instance, there exists an index on the product id (Figure 2, attribute $A$), yet just the price is updated. The pointer in the $VID_{map}$ is updated to point to the $entrypoint$ $X_1$ on page $P5$ while the index is left unchanged. In the worst case the predecessor version $X_0$ has to be fetched after $X_1$, if it did not match the visibility criteria. (i) Data items that are updated without the change of the *key-value* do not require an update on the index structure in SIAS-Chains. (ii) New (most recent) tuple versions, that denote the $entrypoint$ of the data item, will eventually become the only visible version of the data item.

# 5. EVALUATION

We implemented the SIAS-Chains algorithm as a prototype in PostgreSQL [2]. We compare it to the original PostgreSQL code base, utilizing the DBT2 [1] open source implementation of the TPC-C [4] benchmark. Each single run of the TPC-C benchmark is configured with a different amount of warehouses (WHs) and the results are reported in terms of transactional throughput (new order transactions per minute - NOTPM) and in response time (seconds - sec.). We evaluate SIAS-Chains on different storage system configurations: i) a SSD RAID comprised of two SSDs, 4GB RAM and an Intel Core2DUO; ii) a server (Sylt) with an array of six SSDs configured in a stripe RAID, 80GB RAM and two Intel Xeon CPUs. The utilized SSDs are of the type of enterprise class SLC Flash (Intel X25-E 64GB SLC SSD).

## 5.1 I/O Pattern

To visualize the effects of SIAS-Chains on the I/O device we recorded a blocktrace of a testrun and compare it with the I/O blocktrace of SI.

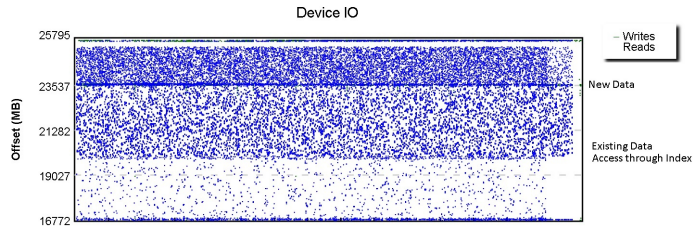


**Figure 3: Blocktrace: SIAS-Chains - SSD - 100 Warehouses - 300 sec. Almost only read access is issued.**

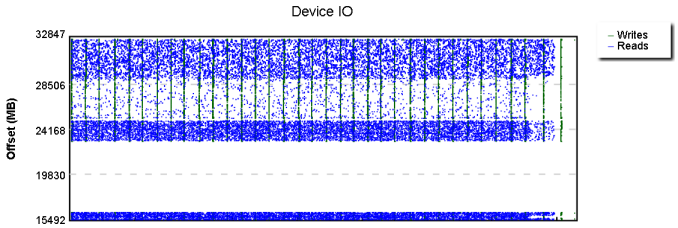


**Figure 4: Blocktrace: SI - SSD - 100 Warehouses - 300 sec. Read and write access is mixed.**

**Table 1: Write Amount (MB) and Reduction (%)**

| Time(sec.) | SI | SIAS-$t1$ | SIAS-$t2$ | Red.-$t1$ | Red.-$t2$ |
|---|---|---|---|---|---|
| 600 | 4369 | 1511 | 130,5 | 65% | 97% |
| 900 | 6488 | 2263 | 193,6 | 65% | 97% |
| 1800 | 12786 | 4473 | 344,65 | 65% | 97% |

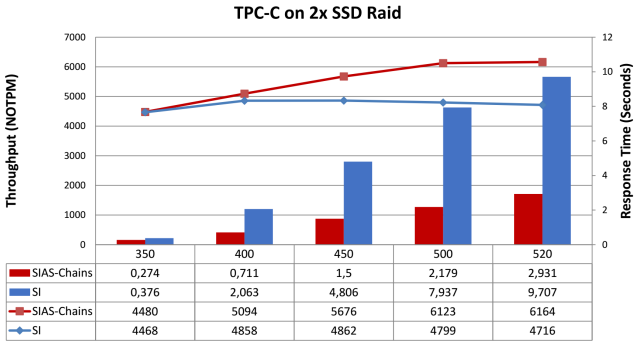


**Figure 5: TPC-C benchmark on a two SSD RAID. Throughput measured in new order transactions per minute.**

In SI write accesses are scattered along the whole relation, causing in-place updates and small random writes. Whereas in SIAS-Chains the write access is streamlined and only appends are issued. Each dot represents a page write of 8kb in each trace.

In SIAS-Chains each relation appends new pages containing newly created tuple versions on a local append region (green dots in the Figures). The scan operation over the $VID_{map}$ leads to a more selective access which creates random read operations.

SI produces a high amount of in-place updates due to the in-place invalidation of older tuple versions. A write operation implies that a page is read first. Pages are scanned and updated in sequence, hence the diagonal pattern.

The amount of write requests is significantly reduced. Table 1 shows the total amount of writes for SIAS-Chains and SI using *blkparse* on a TPC-C dataset of 100 warehouses (WH). This is caused by the append of tuple versions to a page, that is appended and flushed to the storage after a threshold is reached, see Section 5.2 for details. The trace diagrams (Figures 3 and 4) show that recently appended pages are accessed more often than pages that contain cold(er) tuple versions. Recently inserted tuple versions are likely to be cached. Figure 3 shows that under SIAS-Chains data is selectively read, forming a scattered distribution, utilizing the ample I/O parallelism of the Flash storage. Appends to each relation form swimlanes.

## 5.2 Write Reduction

In the evaluation we observe a significant write reduction by the SIAS-Chains algorithm.

The Flash device suffers write overhead from SI caused by the in-place invalidation: if a tuple version is invalidated, only the timestamp (32bit) is updated, while the tuple version data stays unaltered. Nevertheless, the whole page (8KB) has to be updated out-of place on Flash, which eventually leads to a whole physical erase-rewrite operation, entailing mapping overhead and unpredictable performance. SI writes the new version on any (arbitrary) page that contains enough free space, possibly causing another erase-rewrite cycle. This overhead is conceptually avoided in SIAS-Chains, which explains the significant write reduction. SIAS-Chains appends the new version, leaving the old one unchanged and updates the pointer in the in-memory $VID_{map}$.

We measured the amount of write reduction on a larger set of WHs. We recorded the blocktraces and configured DBT2 with a scaling factor of hundred WHs on different runtimes in order to inspect the amount of writes issued by SI and SIAS-Chains as well as to measure the occupied space. Table 1 depicts our findings.

The amount of *write reduction* depends on the filling degree of each appended page, determined by a threshold, directly influencing the amount of *occupied space*. It defines when a new page is physically appended to stable storage. We configured SIAS-Chains with two different thresholds. Threshold $t1$, the default setting of the PostgreSQL background writer process and $t2$ defined by each checkpoint interval (piggy back).

The SIAS-Chains algorithm leads to an immense write reduction. In comparison to SI, SIAS-Chains only issues 3% of the amount of writes and therefore cuts down on 97% of the overall write requests, listed in Table 1 with threshold SIAS-$t2$. This means that the amount of writes issued by SIAS-Chains is 33 times less than the amount that is issued by SI. Even with threshold $t1$ the reduction amounts to 65%. This is extremely beneficial for the Flash storage in terms of write stability, endurance and lifetime.

Tuples of different relations are not stored on the same page and pages that belong to different relations are placed at different location. This reduces contention, e.g. one relation receives more updates/inserts than another [17].

Threshold $t1$ is less suitable for SIAS-Chains: sparsely filled pages are persisted too frequently, leading to a poor overall space consumption, wasted space and a higher amount of write requests.

Configured with $t2$ SIAS-Chains delivers a reduction of the overall space consumption by 12%. Hence pages are filled denser and a smaller number of pages is physically appended.

The optimal threshold for write efficiency is the maximum filling degree of a page - as reported in [17] and [21].

## 5.3 Performance on Flash Storage

Figure 5 depicts the throughput and response times of both approaches on a software stripe RAID comprised of two SSDs. SIAS-Chains achives high throughput with a larger amount of WHs. Through-

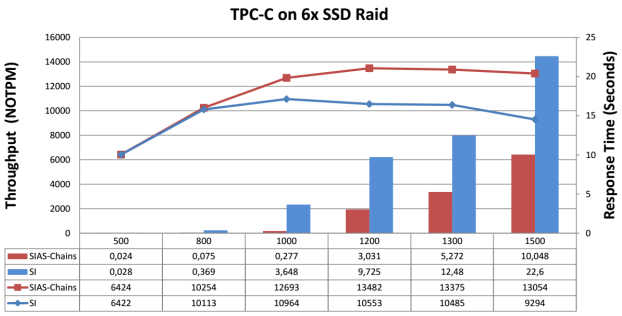


**Figure 6: TPC-C benchmark on six SSDs in software RAID (stripe). Throughput in new order transactions per minute. Throughput in Seconds.**

**Table 2: TPC-C on HDD - Throughput (NOTPM) and Response Time (sec.)**

| Warehouses | 30 | 40 | 50 | 60 | 75 | 100 |
|---|---|---|---|---|---|---|
| SIAS (NOTPM) | 386 | 512 | 642 | 763 | 942 | 727 |
| SI (NOTPM) | 325 | 307 | 279 | 247 | 243 | 204 |
| SIAS (sec.) | 0,031 | 0,05 | 0,2 | 0,3 | 2,1 | 20,35 |
| SI (sec.) | 11,7 | 31,4 | 46 | 65 | 82 | 123 |

put is increased by 30%. SI reaches peak throughput at 450 WHs with a response time of 4.8 sec. delivering 4862 NOTPM. SIAS-Chains reaches the peak with 530 WHs with a response time of 3.3 sec., delivering 6182 NOTPM.

## 5.4 Performance on HDD

With SIAS-Chains we observed clear advantages on HDD (Seagate ST3320613AS S-ATA HDD with 7200rpm).

The caching effect of SIAS-Chains is implicitly confirmed by our performance tests on HDD. SIAS-Chains scales on HDD as long as most reads are cached and improves on SI due to write reduction and append operations. Since less data is written in SIAS-Chains, the HDD has to move less mechanical components for random writes. It is therefore not surprising that SIAS-Chains proves to be beneficial for the HDD as long as the sum of reads and writes (appends) in SIAS-Chains stays below the cost of those in SI (random access costs are symmetric). On HDD SIAS-Chains improves on SI in terms of transactional throughput and especially on response time (better caching). SI exhibits high response times. The system stays responsive below 30 WHs. SIAS-Chains provides a responsive system with up to 75 WHs.

# 6. DISCUSSION

*Recovery.* SIAS-Chains does not affect the MV-DBMS's inherent recovery mechanisms (write-ahead logging and ARIES-style recovery). The threshold delays the time until a single page is allowed to be appended and written out to the storage media, yet WAL is not affected. Currently the SIAS-Chains data structures are only persisted during the shutdown of the DBMS, since all information that is required for a reconstruction is stored on each tuple version (Section 4.1.1).

*Space Reclamation.* The basic concept in MV-DBMSs is to reclaim space on the append storage using a garbage collection (GC) mechanism which: (i) finds a victim page that is chosen to be garbage collected, (ii) re-inserts live (visible) tuple versions and (iii) discards dead (invisible) tuple versions of that page.

GC mechanisms have to erase pages on the SSD, yet this is a deterministic process, triggered by the MV-DBMS and does not rely on the device's inherent mechanisms. Integrating the append storage GC into the MV-DBMS avoids unpredictable performance outliers of the Flash storage media, caused by background processes on the device. This transfers yet more control over the Flash storage into the MV-DBMS, as in the approach reported in [22].

*Flash Endurance.* Since Flash storage is prone to wear and has a limited lifetime, it is crucial to provide appropriate I/O patterns in order to improve the endurance. Growing Flash capacities for SSDs entail the trend to larger erase units, since larger sizes allow the use of smaller mapping tables on the SSD's controller. Since wear on the device is measured using the average amount of erases, avoidance of small updates, such as timestamp related meta-information becomes more important. The I/O pattern, as created by SIAS-Chains, suggests an increased endurance of the Flash memories. SIAS-Chains significantly reduces the write amount and the required space on the device. In-place updates are avoided, new pages are written (appended) in monotonously increasing order.

# 7. CONCLUSION

The design, architecture, algorithms and optimizations in MV-DBMSs that are based on characteristics of legacy storage media have to be reconsidered. The full potential of asymmetric storage technologies such as Flash can only be leveraged by MV-DBMSs that are aware of their characteristics and exploit their properties.

With SIAS-Chains we propose a *Flash-aware* MV-DBMS design that contributes: (a) a new paradigm to version management; (b) a new version invalidation scheme; (c) a multi-versioned indexing scheme; (d) the combination of multi-version concurrency control with append storage in tuple granularity; (e) extending it with multi-version indexing, simplified buffer management and read optimizations that leverage the properties of the aforementioned combination. SIAS-Chains algorithmically avoids small in-place updates, caused by in-place invalidation and converts them into appends. Every modification operation is executed as an append and recently inserted tuple versions are co-located.

We implemented SIAS-Chains in the PostgreSQL and evaluated it under TPC-C using DBT2 on different Flash storage media as well as on traditional HDD storage. We observe: (i) higher transactional throughput; (ii) significantly lower response times (orders of magnitude); (iii) significant write reduction; (iv) reduced space consumption; (v) higher amount of tolerable load.

The I/O pattern suggests an increased endurance of the Flash memories. We also report benefits on HDD, especially in terms of lower latency and higher transactional throughput.

# 8. REFERENCES

[1] Dbt2 tpc-c benchmark, http://sourceforge.net/projects/osdldbt/files/dbt2/.

[2] Postgresql mvdbms, http://www.postgresql.org/.

[3] SIAS-V Demo Video, http://dblab.reutlingen-university.de/tl_files/downloads/SIAS_V-InAction.mp4.

[4] Tpc benchmark c standard specification, http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[5] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX*, 2008.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, May 1995.

[7] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - A transactional record manager for shared flash. In *CIDR*, pages 9–20. www.crdrdb.org, 2011.

[8] B. Bhattacharjee, M. Canim, M. Hamedani, K. Ross, and A. Storm. Supporting transient snapshot with coordinated/uncoordinated commit protocol. US Patent 14/748,438, 2016.

[9] P. Bober and M. Carey. On mixing queries and transactions via multiversion locking. In *Proc. ICDE'92*.

[10] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD*, SIGMOD '08, New York, NY, USA. ACM.

[11] M. J. Carey and W. A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Trans. on Computer Sys.*, 4(4):338, Nov. 1986.

[12] 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 *Proc. SIGMOD'82*.

[13] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. SIGMETRICS '09. ACM.

[14] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql servers memory-optimized oltp engine. Sigmod, 2013.

[15] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The sap hana database–an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[16] R. Gottstein, T. Peter, I. Petrov, and A. Buchmann. Sias-v in action: Snapshot isolation append storage - vectors on flash. In *17th International Conference on Extending Database Technology (EDBT)*, 2014.

[17] R. Gottstein, I. Petrov, and A. Buchmann. Append storage in multi-version databases on Flash. In *Proc. BNCOD 2013*.

[18] R. Gottstein, I. Petrov, and A. Buchmann. Si-cv: Snapshot isolation with co-located versions. In *Proc. TPC-TC*. 2012.

[19] R. Gottstein, I. Petrov, and A. Buchmann. Aspects of Append-Based Database Storage Management on Flash Memories. In *DBKDA 2013*, pages 116–120, 2013.

[20] R. Gottstein, I. Petrov, and A. Buchmann. Multi-version databases on flash: Append storage and access paths. *IJAS*, 6(3&4):321–328, 2013.

[21] R. Gottstein, I. Petrov, and A. Buchmann. Read Optimisations for Append Storage on Flash. 2013.

[22] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann. Noftl: Database systems on ftl-less flash storage. *Proc. VLDB Endow.*, 6(12):1278–1281, Aug. 2013.

[23] A. Kemper and T. Neumann. Hyper - hybrid oltp&olap high performance database system. Technical report, TU München, 2010.

[24] 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 *(PVLDB), Vol. 5*, number 1 in 5, 2011.

[25] D. Majumdar. A quick survey of multiversion concurrency algorithms, 2006.

[26] I. Petrov, R. Gottstein, and S. Hardock. Dbms on modern storage hardware. In *Proc. ICDE*. IEEE, 2015.

[27] I. Petrov, R. Gottstein, T. Ivanov, D. Bausch, and A. P. Buchmann. Page size selection for OLTP databases on SSD storage. *JIDM*, 2(1):11–18, 2011.

[28] D. R. Ports and K. Grittner. Serializable snapshot isolation in postgresql. *Proceedings of the VLDB Endowment '12*, (12-5).

[29] M. Rosenblum. The design and implementation of a log-structured file system. Report ucb/csd 92/696, ph.d thesis, U.C., Berkeley, June 1992.

[30] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee. Exploiting ssds in operational multiversion databases. *VLDB*, 25(5):651–672, Oct. 2016.

[31] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, 6th Edition*. McGraw-Hill, 2011.

[32] R. Stoica and A. Ailamaki. Improving flash write performance by using update frequency. *Proc. VLDB Endow.*, 6(9):733–744, July 2013.

[33] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In *Proc. DaMoN 2009*, pages 9–14, 2009.

[34] M. Stonebraker, L. A. Rowe, and M. Hirohama. The implementation of postgres. *IEEE TKDE '19*, 2(1):125.

[35] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[36] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB*, 2017.