Impact of new storage technologies on an OLTP DBMS, its architecture and algorithms

Auswirkung neuer Speichertechnologien auf ein OLTP DBMS, dessen Architektur und Algorithmen

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte Dissertation zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
von Robert Gottstein, M.Sc. aus Darmstadt

Tag der Einreichung: 4. Februar 2016, Tag der Prüfung: 23. März 2016
Darmstadt 2016 — D 17

1. Gutachten: Alejandro Buchmann
2. Gutachten: Ilia Petrov

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Databases and Distributed Systems

Contents

1 Introduction 9
1.1 Motivation . . . 9
1.2 Inertia of the DBMS . . . 13
1.3 Workload . . . 15
1.4 Transactional Model . . . 15
1.5 Storage Management . . . 15
1.6 Access Methods - Searching and Indexing . . . 16
1.7 Buffer Management . . . 16
1.8 Snapshot Isolation Append Storage . . . 17
1.9 Research Hypothesis and Contributions . . . 17
1.10 Structure of the Dissertation . . . 19

2 New Storage Technologies 21
2.1 Flash Memory . . . 21
2.2 Solid State Drives . . . 24
2.3 Flash Endurance . . . 28
2.4 Hardware Specifications . . . 31
2.5 SSD RAID Configuration . . . 34
2.5.1 Sylt Testbed . . . 35

3 Transaction Management 39
3.1 Multi Version Concurrency Control . . . 39
3.2 MV-DBMS . . . 40
3.3 Multi Version Concurrency Control on SSD . . . 40

4 Version Management 43
4.1 Invalidation Model . . . 44
4.1.1 One place invalidation . . . 44
4.1.2 Two place invalidation . . . 45
4.2 Snapshot Isolation . . . 45
4.2.1 Snapshot Isolation - Introduction . . . 45
4.2.2 Snapshot Isolation - Algorithm . . . 49
4.3 SIAS: Snapshot Isolation Append Storage . . . 51
4.4 SIAS-Vectors . . . 52
4.4.1 SIAS-Vectors - Data Structures . . . 52
4.4.2 SIAS-Vectors - Access Paths . . . 54
4.4.3 SIAS-Vectors - Discussion and Evaluation . . . 56
4.5 SIAS-Chains . . . 65
4.5.1 SIAS-Chains - Algorithm . . . 65
4.5.2 SIAS-Chains - On-Tuple Information . . . 66
4.5.3 SIAS-Chains - Data Structures . . . 67

4.5.4 SIAS-Chains - Access Methods Insert - Update - Delete 71
4.5.5 SIAS-Chains - Discussion and Evaluation 73
4.6 SIAS Garbage Collection 86
4.7 SIAS Buffer Management 87
4.8 SIAS Recovery 88
4.9 SIAS Discussion - Worst Case Scenario 88

5 Access Methods - Searching and Indexing 89
5.1 SIAS-Chains: Scan - Introduction 89
5.1.1 SIAS-Chains - VID_{map} Scan 89
5.1.2 SIAS-Chains - Full Relational Scan 90
5.2 SIAS-Chains: Indexing - Introduction 91
5.2.1 SIAS-Chains - Indexing 91

6 Storage Management 93
6.1 Append Storage - Introduction 93
6.2 Append Granularity 94
6.2.1 Page Append LbSM 96
6.2.2 Tuple Append LbSM 96
6.3 Data Placement and Layout 96
6.3.1 Global Region 97
6.3.2 Local Region 97
6.4 LbSM Evaluation 98
6.4.1 Simulation. 99
6.4.2 Validation 101
6.4.3 Results 103
6.5 Read Optimizations 114
6.5.1 Index-Snippet. 115
6.5.2 Ordered/Unordered Append Log. 118

7 Buffer Management 129
7.1 Related Buffer Management Approaches 130
7.1.1 CFLRU 130
7.1.2 CFDC 130
7.1.3 FOR 130
7.1.4 DULO 131
7.1.5 BPLRU 131
7.2 Flash Based Adaptive Replacement Cache 132
7.2.1 FBARC - Algorithm 133
7.3 FBARC: Instrumentation and Experiments 138
7.3.1 Raw Trace 141
7.3.2 Workload and Trace Characterization 141
7.4 FBARC: Experimental Evaluation 145
7.4.1 CPU Time 145
7.4.2 I-O Time 146
7.4.3 Overall Time 147

7.4.4 Scan Resistance . . . 148
7.4.5 FBARC Cluster Sizes . . . 149
7.4.6 Behavior on HDD . . . 150
7.4.7 FBARC Summary and Conclusion . . . 150

8 Related Work 155

8.1 Multi Version Databases . . . 155
8.1.1 Snapshot Isolation . . . 155
8.1.2 Indexing in MV-DBMS . . . 156
8.2 Temporal Databases . . . 156
8.3 In-Memory Databases . . . 156
8.4 Log Based Storage Management . . . 157
8.5 Snapshot Isolation Append Storage . . . 157
8.6 Flash Storage . . . 158
8.7 Flash Translation Layer . . . 158
8.8 Buffer Management . . . 159

9 Summary and Future Work 161

9.1 Summary and Contributions . . . 161
9.2 Future Work . . . 164
9.2.1 Garbage Collection . . . 164
9.2.2 Thinking Beyond the Block Interface . . . 164
9.2.3 Compression of Sorted Runs . . . 164

List of Figures

1.1 Memory Hierarchy 10
1.2 Invalidation in SIAS-Chains and SI 13
1.3 MV-DBMS Architecture with focus on transactional processing 14

2.1 NAND Flash Layout 23
2.2 Flash SSD Layout 25
2.3 Flash Translation Layer Mapping Scheme 26
2.4 Throughput on different block sizes 27
2.5 SSD Latency Outliers 29
2.6 SSD state dependency 30
2.7 Intel X25-E SSD 32
2.8 Mixed Workload Performance with Caching 33
2.9 Mixed Workload Performance without Caching 33
2.10 Sylt server installed in server rack 35
2.11 Sylt server layout 36
2.12 Sylt server read performance (IOPS) raw 37
2.13 Sylt server read performance (ms) raw 37
2.14 Sylt server write performance (IOPS) raw 38
2.15 Sylt server write performance (ms) raw 38

4.1 Example history H 46
4.2 History H - relational view 47
4.3 History H - Snapshot Isolation invalidation scheme 47
4.4 Version scheme related write amplification 49
4.5 SIAS Vectors data structures 52
4.6 SIAS-Vectors and SI on-tuple information 53
4.7 History H - SIAS-Vectors invalidation scheme 54
4.8 SIAS tuple append related write reduction 55
4.9 SIAS-Vectors evaluation process 56
4.10 SIAS-Vectors write pattern - TPC-C customer relation 57
4.11 SIAS-Vectors write pattern - TPC-C stock relation 58
4.12 SI write pattern - TPC-C customer relation 58
4.13 SI write pattern - TPC-C customer and stock relation 59
4.14 Blocktrace SI 59
4.15 Blocktrace SIAS-Chains 60
4.16 SIAS-Vectors one tuple update microbenchmark 61
4.17 SIAS-Vectors one tuple update microbenchmark 61
4.18 One tuple update micro benchmark on HDD 62
4.19 One tuple update micro benchmark on HDD 62
4.20 SIAS-Vectors length of the version chain 63
4.21 SIAS-Vectors length of the version chain 64
4.22 SIAS-Chains invalidation scheme - History H 66

4.23 SIAS-Chains and SI on-tuple version information 67
4.24 SIAS-Chains relational view 68
4.25 SIAS-Chains datastructure hash buckets 70
4.26 SIAS-Chains datastructure - B-tree 71
4.27 Blocktrace SIAS-Chains USB Flash 74
4.28 Blockstrace SI USB Flash 75
4.29 Blocktrace SIAS-Chains SSD Flash 75
4.30 Blocktrace SI SSD Flash 76
4.31 SIAS-Chains vs. SI TPC-C throughput on USB Flash 78
4.32 SIAS-Chains vs. SI response time on USB Flash 79
4.33 SIAS-Chains vs. SI TPC-C throughput on SSD 79
4.34 SIAS-Chains vs. SI TPC-C response time on SSD 80
4.35 SIAS-Chains vs. SI TPC-C throughput on SSD RAID 81
4.36 SIAS-Chains vs. SI TPC-C response time on SSD RAID 81
4.37 SIAS-Chains vs. SI TPC-C throughput on Sylt 82
4.38 SIAS-Chains vs. SI TPC-C response time on Sylt 82
4.39 SIAS-Chains TPC-C response time and throughput on USB Flash 83
4.40 SIAS-Chains TPC-C response time and throughput on SSD 84
4.41 SIAS-Chains TPC-C response time and throughput on SSD RAID 84
4.42 SIAS-Chains vs. SI TPC-C throughput on HDD 85
4.43 SIAS-Chains vs. SI - Response Time on HDD 85
4.44 SIAS-Chains garbage collection concept 86

5.1 SIAS-Chains indexing concept 92

6.1 Storage Managers - Overview 95
6.2 Global vs. Local append regions 97
6.3 Evaluated storage managers - Overview 99
6.4 LbSM evaluation process 100
6.5 Blocktrace SIAS-Chains on USB 102
6.6 Blocktrace SIAS-Chains on SSD 102
6.7 LbSM blocktrace on SSD 103
6.8 LbSM and I/O parallelism - IOPS - Trace I 106
6.9 LbSM and I/O parallelism - IOPS - Trace I 106
6.10 LbSM and I/O parallelism - Runtime - Trace I 107
6.11 LbSM and I/O parallelism - Runtime - Trace I 107
6.12 LbSM and I/O parallelism - IOPS - Trace II 108
6.13 LbSM and I/O parallelism - Runtime - Trace II 108
6.14 Blocktrace page append 110
6.15 Blocktrace page append 110
6.16 Blocktrace global append region 111
6.17 Blocktrace local append regions 111
6.18 Blocktrace in-place storage management 112
6.19 Blocktrace SIAS tuple LbSM 112
6.20 Blocktrace SI vs. SIAS-Chains local append - write 113
6.21 Blocktrace SI vs. SIAS-Chains local append - read 113

6.22 Index Snippet 117
6.23 Ordered page and tuple append storage managers 119
6.24 Blocktrace unordered append 124
6.25 Blocktrace ordered append 125
6.26 Blocktrace ordered append 126
6.27 Blocktrace unordered append 127

7.1 Memory Hierarchy 133
7.2 FBARC concept 134
7.3 FBARC evaluation process 140
7.4 FBARC evaluation Trace H 142
7.5 FBARC evaluation Trace B 142
7.6 FBARC evaluation Trace C 142
7.7 FBARC evaluation Trace C 142
7.8 Buffer manager hitrate comparison 144
7.9 Buffer management CPU overhead 146
7.10 FBARC overall speedup Trace C 147
7.11 FBARC overall speedup Trace C 148
7.12 Influence of scan resistance on hitrate 149
7.13 Buffer management on HDD I/O time 150
7.14 Buffer management relative performance to ARC 151
7.15 Trace H scatter diagram 151
7.16 Trace C scatter diagram 152
7.17 Trace H scatter diagram - delivery relation only 152
7.18 Trace B scatter diagram 153

9.1 Benefits of SIAS in a nutshell 163

List of Tables

2.1 SSD vendor specifications . . . 22
2.2 Single drive performance figures under Windows 2008 Server . . . 31
2.3 Single drive performance figures under Ubuntu Linux Server . . . 32
2.4 Sylt specification. . . . 35
2.5 Sylt baseline performance under Linux. . . . 36

4.1 SIAS-Chains: On-Tuple Information . . . 67
4.2 SIAS-Chains write reduction on TPC-C workload . . . 77

6.1 Comparison: ordered page LbSM and ordered tuple LbSM . . . 120

7.1 Average I/O Latencies for the Intel X25-E SSD and the Hitachi HDS72161 HDD. . . 140
7.2 Amount of read data (KB) in Trace C . . . 146
7.3 Amount of written data (KB) in Trace C . . . 147
7.4 Influence of cluster size on hitrate, CPU and overall time . . . 149

1 Introduction

Contents

1.1 Motivation 9
1.2 Inertia of the DBMS 13
1.3 Workload 15
1.4 Transactional Model 15
1.5 Storage Management 15
1.6 Access Methods - Searching and Indexing 16
1.7 Buffer Management 16
1.8 Snapshot Isolation Append Storage 17
1.9 Research Hypothesis and Contributions 17
1.10 Structure of the Dissertation 19

1.1 Motivation

Traditional storage such as Hard Disk Drives (HDDs) reached physical limits regarding latency and I/O performance. Scalability and further performance improvement is limited by physical boundaries: the mechanical/material characteristics of moving parts. With this traditional storage an access gap has been formed. Data-intensive systems such as database management systems (DBMSs) that handle on line transactional load (OLTP) workloads are I/O bound and demand scalable, fast access storage media. Modern DBMSs are designed on the performance characteristics and device properties of HDD legacy hardware storage systems. Dated at least three decades back, several optimizations have been integrated to utilize traditional hardware and to compensate for their weak spots, such as their high latency.

Aiming to reduce I/O waiting time, large buffer and page sizes have been employed. Algorithms are optimized for clustering, and streaming of data rather than random access. The buffer management, for example, is designed and optimized to *buffer* the impact of the access gap. The access gap is a technological property of traditional memory hierarchies: accessing data in a buffer resident page takes less than hundred nanoseconds whereas accessing disk-based data is in the lower millisecond range (Figure 1.1).

New developments in hardware storage technology introduce fundamentally different performance characteristics and device properties compared to traditional (*legacy*) storage hardware (Chapter 2). Figure 1.1 depicts the memory hierarchy. Storage technologies such as Flash and Non Volatile Memories (NVMs) are capable of narrowing down the access gap and satisfy the urgent need of modern DBMSs. These storage technologies are *asymmetric* in terms of their read and write performance, they read much faster than they write. However their asymmetric read and write behavior introduces new problems.

Existing DBMSs are not aware of the underlying asymmetric storage technologies. A plain replacement with new storage technologies delivers only small performance benefits, while the

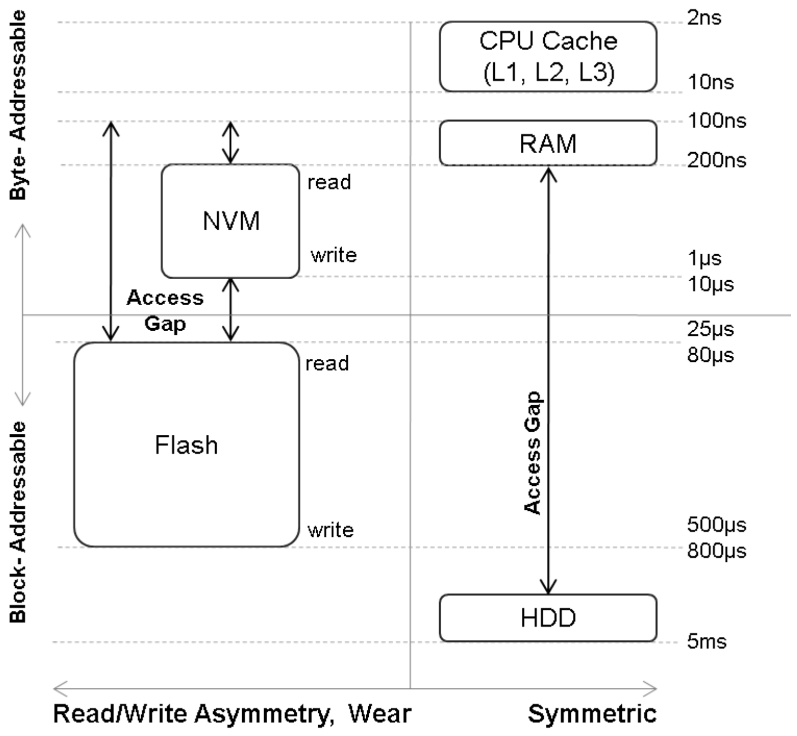


Figure 1.1: Memory Hierarchy and Access Gap. Asymmetric storage technologies fill in the access gap between symmetric storage. While the access gap shrinks it becomes operation dependent, impacting existing algorithms and DBMS design assumptions.

main part of the potential can't be realized. For backwards compatibility these storage technologies implement legacy hardware interfaces, the so called *block interface* and software layers. To mask the properties and read/write asymmetry, they support legacy access methods by adding additional software layers and mappings, which lead to complexity that artificially limits their performance. This is unnecessary since a *Flash-aware* instrumentation of the new Flash storage reduces overheads, removes additional *compatibility* layers and simplifies the DBMS architecture as well as the storage stack.

In recent years Flash based storage systems for DBMSs became a standard technology. They deeply impact the DBMS architecture and algorithms. This is not a simple question of *"setting parameters right"*: the overall DBMS architecture and inherent algorithms have to be re-evaluated and re-designed. The Flash storage technology is based on electronic chips and does not require moving parts.

In comparison with traditional Hard Disk Drives (HDDs), Flash Solid State Drives (SSDs) have these key differences (a detailed description of the properties is given in Chapter 2):

- Asymmetric read/write performance: write approximately 10x slower than read;
- No overwrite: explicit erase operation (approx. 10x slower than write);
- No mechanical latency;
- Inherent parallelism;
- Limited Durability: *erase-cycles* wear out the Flash memory cells;

- Low energy consumption and better performance per watt.

In order to mask several of these properties, the manufacturers of SSDs employ large volatile caches (SD-RAM) and complex management processes that are implemented on the device in a black box design. Although this fosters operational availability and interchangeability, this can only be seen as a temporal solution: complexity and cost of the SSDs is increased. State dependencies as well as unpredictable I/O and latency behavior (outliers) are direct consequences. The asymmetry is exposed as soon as the volatile cache is filled. Complex, time consuming erase operations are delayed by a remapping of updated pages, which requires over-provisioning and complex garbage collection (GC) mechanisms. The GC interferes with foreground I/O requests, causing noticeable performance fluctuations and unpredictability.

These inherently different properties result in a deep discrepancy in the design of DBMS architecture, algorithms and optimizations [39]. Some examples [111] are:

- on HDDs a *clustering* is maintained to optimize for follow-up read sequential accesses, yet in-place updates are necessary to maintain the clustering and to optimize for follow-up read accesses, but have to be avoided on SSDs as they result in costly erase-cycles which hamper performance and decrease endurance;
- Large page sizes and blocked I/O is preferred on HDDs, since more data is transfered per I/O and I/O waiting time can be reduced. *Cache pollution* can occur, where the cache is filled with unused data that has been transfered during the I/O;
- SSDs are capable of fast, parallel random reads and yield low latency on small page sizes, which decreases the effect of *cache pollution* [91]. This also questions the trend towards larger pages sizes;
- The buffer management for symmetric storage is mainly optimized for hitrate to reduce the amount of random reads (re-fetching of pages), where asymmetric storage is fast on random and on sequential reads.

Overall, the developments of the past years aim at the reduction of the access gap between symmetric storage and symmetric memory. A gap which now can be filled by the asymmetric storage technologies such as Flash.

Modern DBMSs are well-developed systems and in principle capable of working with asymmetric storage technologies as a mere replacement, yet they fail to exploit their key properties. This leads to the situation where huge performance potential is lying idle and durability of the storage media is shortened which ultimately leads to higher costs.

This work is a remedy for those shortcomings, making the DBMS aware of the underlying asymmetric Flash storage and questioning existing multi-version DBMS (MV-DBMS) architecture, algorithms and optimizations. We exploit the performance potential of the asymmetric Flash storage and increase its durability. A re-evaluation and redesign of components within the DBMS is necessary, inevitably leading to a redesign of the whole DBMS. Without such a redesign, the DBMS software stack will become the new I/O bottleneck.

In this work we argue that the combination of the MV-DBMS, multi-versioning concurrency control (MVCC) and append/log-based storage management (LbSM) on Flash storage delivers optimal performance figures which are needed to satisfy the urgent demand in scalable performance for modern DBMSs. The concepts need to be well integrated into a sound *Flash-aware* system.

This demands a reconsideration of:

- the transactional multi-versioning invalidation model;
- searching and indexing schemes (access paths);
- the storage management and
- the buffer management.

This thesis describes a redesign of those concepts and further combines them to form an integrated, sound MV-DBMS by introducing the *Snapshot Isolation Append Storage (SIAS)* approach. In the following the concepts are introduced, focusing on OLTP workloads. The following Section 1.2 provides an overview of MV-DBMSs.

1.2 Inertia of the DBMS

A first step of this work is the re-evaluation of the inertia, architecture and algorithms, of existing DBMSs. The transactional model of a Multi Versioning DBMS (MV-DBMS) seems to be a good match for the Flash properties. Figure 1.3 depicts the general architecture of a MV-DBMS. Modules adapted as part of this thesis are shaded grey. Transaction processing and the transactional model of the DBMS are directly influenced by hardware-specific properties of the underlying physical storage. Properties such as the read/write asymmetry, slow in-place updates and write durability need to be addressed while I/O parallelism and low I/O latency have to be leveraged.

MV-DBMSs can benefit from the fast reads, the organization of the underlying storage, low latencies and the parallelism. Out-of-place updates are well suited for the organization of the data on Flash, mainly addressing issues of slow overwrites, asymmetry and mixed load performance.

Parallelism is complemented by the MV-DBMS' capability to support lock-free reads. Read requests of concurrently running transactions can be executed in parallel on Flash.

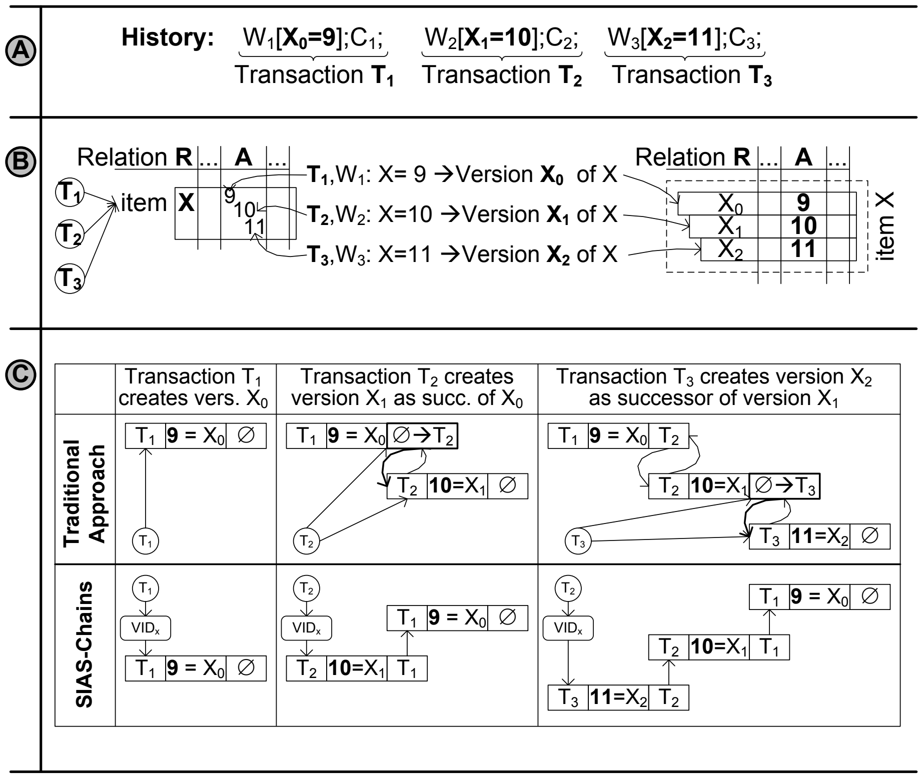

A
History: W1[X0=9];C1; W2[X1=10];C2; W3[X2=11];C3;
Transaction T1
Transaction T2
Transaction T3
B
Relation R ... A ...
item X
T1,W1: X= 9 → Version X0 of X
T2,W2: X=10 → Version X1 of X
T3,W3: X=11 → Version X2 of X
item X
C
Transaction T1 creates vers. X0
Transaction T2 creates version X1 as succ. of X0
Transaction T3 creates version X2 as successor of version X1
Traditional Approach
SIAS-Chains
VIDx

Figure 1.2: Invalidation in SIAS-Chains and the traditional approach (SI): (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.

Ideally a MV-DBMS that utilizes the high I/O parallelism of Flash, enables concurrent reading and parallel writing, while the multi-versioning transactional model enables concurrent transaction processing. Since modern MV-DBMSs are designed for the properties of HDDs, they are not capable to exploit the parallelism of the Flash storage. The critical point of the MV-DBMS is the inherent version organization. A version organization scheme that implements an *in-place invalidation* model can't leverage the conceptual advantages of the MV-DBMS. In-place invalidation models create (small) in-place-updates which render conceptual benefits inaccessible. In order to exploit the versioning scheme, the invalidation model has to be changed to an out-of-place invalidation model. Physical out-of-place updates need to be enabled. This also complements append only storage management on the algorithmic level. Figure 1.2 shows an example of the version invalidation scheme of the traditional approach (Section 4.2) that utilizes the two-place invalidation scheme (Section 4.1.2). Figure 1.2 also depicts the Snapshot Isolation Append Storage approach (SIAS), which is explained in Section 4.3. SIAS employs one-place invalidation (Section 4.1.1) and enables out-of-place updates.

The storage management is the second critical component of the MV-DBMS. It determines *how* data is read from and written to the storage. Overwrites on the Flash storage are more costly than writing to *fresh or erased* regions on the Flash storage. Streamlining of write access, avoiding random writes and in-place updates determines the choice for an append based storage management. Historically such approaches are not integrated into the MV-DBMS since they are only beneficial in special cases on symmetric storage, e.g. on streamlined writes. Mostly they are implemented as an additional layer on top of the file system or device driver.

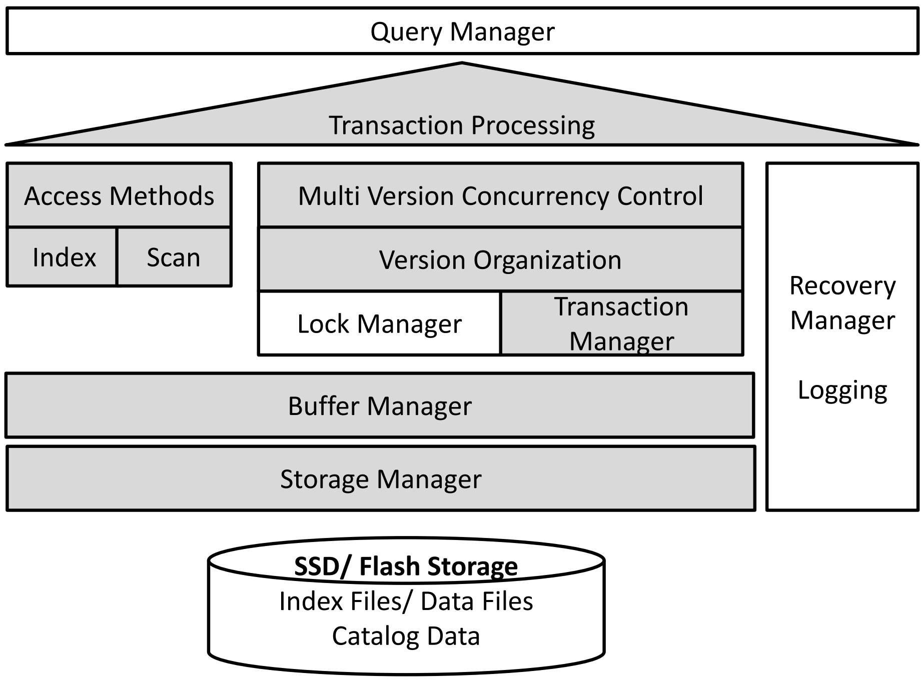


Figure 1.3: MV-DBMS Architecture with focus on transactional processing. Modules adapted as part of this thesis are shaded gray.

1.3 Workload

In order to design and architect a sound *Flash-aware* MV-DBMS the workload must be considered: this work focuses on *On-Line Transactional Processing* (OLTP), which can hugely benefit from Flash storage. OLTP mainly creates short running transactions that execute multiple reads, small updates and inserts [113]. These are operations that demand highly concurrent executions and parallel access. Data skew can be observed where approx. 80% of all accesses are issued on only 20% of the data, hence mixed workload and hot-spot regions emerge.

This work focuses on OLTP loads, as which are modeled by the TPC-C benchmark [113]. An overview of the benchmark is provided in the related work (Chapter 8). OLTP workload, as created by the TPC-C benchmark, mainly focuses on short running transactions, creating a transactional mix of 30% (small) update and 70% reading requests. This “mostly read” workload is an ideal case for an MV-DBMS with underlying Flash storage that inherently offers parallel read access with low latencies. In principle the algorithms presented in this work are also suitable for other mixed workloads.

1.4 Transactional Model

The transactional model of MV-DBMSs offers conceptual advantages for Flash storage: lock-free read access complements parallelism and conceptual out-of-place updates alleviate the issue of in-place-updates. In comparison: transactional models of single version DBMS rely on in-place updates and implement blocking approaches to concurrency control based on locking, hence MV-DBMSs offer a higher grade of concurrency. The transactional model is further analyzed in Chapter 3.

Modern MV-DBMSs follow an invalidation model that relies on *in-place invalidation* of tuple versions, effectively locking tuple versions on updates and creating in-place-updates in order to store visibility information. These properties create influence on the storage management, making assumptions on tuple version placement, updates, etc. Version-organization is further described in Chapter 4. In order to properly utilize the underlying Flash storage the invalidation model has to be changed to an out-of-place invalidation model. The higher level of concurrency must be propagated to the Flash storage in order to exploit parallelism. Propagating parallelism to the Flash storage and avoiding the need for in-place invalidation becomes a key issue which is addressed in this work.

1.5 Storage Management

Since Flash storage hugely impacts existing storage management approaches it emphasizes the need for data placement strategies. It is not only performance critical but also vital for endurance and finally the cost of the devices.

Reading access becomes a less critical component, whereas write access patterns must be optimized. A major issue is the endurance of Flash memories which can only be improved by a significant reduction of the write amplification: in-place updates have to be avoided, especially those that are caused by the invalidation model (Section 4.1); mixed load needs to be reduced and random writes have to be avoided (especially those in small granularities). In addition to these write issues, utilizing parallelism and low latency reads becomes a high priority task.

The asymmetry of Flash storage substantiates these assumptions: Flash reads as fast in random as in sequential, therefore the cost of maintaining a physical clustering (in-place updates, overwrites and re-organizations) stands in no relation to the benefits (details in Section2) .

Shown in [109] append storage (page granularity) is beneficial for Flash. In this work we argue that append storage is not only beneficial for Flash, but for any new storage media that featuring asymmetric read/write speed, no support of direct in-place updates and high parallelism. Traditional log-based storage managers (LbSMs) append in the granularity of pages and are implemented as an additional layer *outside* the database [109].

The integration of the append operation into the MV-DBMS enables appends in small granularities of tuple versions instead of pages. This reduces write amplification and the amount of data that has to be written to Flash storage, compared to traditional LbSM approaches, as shown in [35] and demonstrated in [33]. Chapter 6 investigates approaches to LbSM. The integration of the tuple granularity LbSM into the MV-DBMS is described in detail in Section 4.3.

In all these approaches the database is separated from the (recovery-) log. This is further described in Section 6.1.

Integrating the LbSM into the MV-DBMS, involving a change of the transactional model and invalidation scheme, introduces new challenges to access paths and auxiliary structures.

1.6 Access Methods - Searching and Indexing

Storing versions of tuples in a LbSM requires efficient access paths. The change of the invalidation paradigm, versioning and appending tuples instead of maintaining a clustering emphasizes the need for efficient searching and indexing. Access paths in a MV-DBMS deliver access to data items that match a certain (search-) criterion and in addition the versioning has to be managed. Traditional index structures are unaware of the versioned data underneath and the underlying asymmetric storage. They are *version oblivious* and treat each tuple version as a unique data item. Traditional MV-DBMS separately handle visibility of the versions and need to *check and filter* the result set that an index structure returns on a query, since the result set may contain different tuple versions of the same data item. This also involves fetching each indexed version and checking it for visibility. Furthermore on updates of data items new versions are created which have to be inserted into the index, even if the indexed value has not changed. The old index entry has to be updated or removed, which entails index management overhead (e.g. $B^{+}tree$ rotations) resulting in suboptimal I/O patterns. Making the index aware of the versioned data underneath and the underlying asymmetric storage can reduce the otherwise incurred overhead of the version management. For instance, requesting and fetching versions that are not visible involves probing and filtering that has to be provided by the MV-DBMS. Suboptimal access patterns can be avoided, also reported in [37] and [35].

Section 5.2 describes the version conscious SIAS index structure capable of creating selective I/O access to the Flash storage.

Making the access paths aware of the underlying Flash storage and versioning scheme further creates new challenges for the buffer management (see Chapter 7).

1.7 Buffer Management

Buffer Management is a vital part of modern DBMS. Historically this key task is to minimize the effect of the access gap between memory and storage (disk). *Important* data has to be kept near the processing unit. Traditional buffer management algorithms optimized for HDDs optimize for hitrate, based upon metrics such as frequency and recency. Designed for symmetric disk storage they are not *aware* that on asymmetric storage the cost of one dirty-page eviction can be multiple times higher than the cost of (re-)fetching individual pages. This shows that utilizing only hitrate as a metric for an eviction decision is not sufficient on Flash storage any more. A Flash-aware buffer management strategy also has to consider *spatial locality* of dirty pages and needs to treat the costs of reading/writing a page as two different values.

The Flash-based Adaptive Replacement Cache (FBARC) [29] is based on the adaptive replacement cache [77] and focuses on the importance of write clustering, forming sequential or semi-sequential write access patterns. This approach is designed to optimize for in-place update DBMSs, requiring less invasive changes of the DBMS.

The SIAS approach simplifies the buffer management. It applies *write retention* to pages that are appended after the page is completely filled or a threshold is reached. SIAS buffer management is described in Section 4.7 and FBARC is contained in Chapter 7.

1.8 Snapshot Isolation Append Storage

The introduced concepts need to be well integrated into the DBMS in order to form a sound *Flash-aware* system, this lead to the Snapshot Isolation Append Storage (SIAS). This thesis describes the Snapshot Isolation Append Storage (SIAS) approach, a *Flash-friendly* invalidation model and a change in the paradigm of the version organization of existing MV-DBMSs. SIAS implements an out-of-place invalidation model and the transactional model introduces the paradigm of addressing all tuple versions of a data item as a unique entity by augmenting them with a logical Virtual Identifier (VID). Visibility information about invalidation of a tuple version is not stored on the tuple version but is coded within the version chain.

Versions of single relations are appended to new pages, which are appended to the Flash storage as soon as a threshold is reached, e.g. when a page is completely filled. In-Place invalidation, physical overwrites and costly remapping operations on the Flash storage are avoided and read-access parallelism is propagated to the Flash storage, as demonstrated in [33].

The SIAS algorithm introduces Flash-aware access paths and auxiliary structures that are conscious of the versioned data underneath, using the data item's VID as the indexed value. Thus more selective I/O access is possible.

A separate clustering to sequentialize writes is not maintained, since the append operation already writes in sequence (as a DB log). In Section 6.1 the inter-page vs. page only sorting of tuple versions is evaluated and compared to a sorted clustering of pages that are appended to the MV-DBMS storage media [38].

1.9 Research Hypothesis and Contributions

Existing assumptions on design and architecture of modern MV-DBMSs are questioned and revised. The *Snapshot Isolation Append Storage* (SIAS) approach subsumes changes on the algorithmic and architectural design of the MV-DBMS with the focus on asymmetric Flash storage. The contributions of this dissertation are:

- Storage Management: single-/multi-version schemes that employ in-place update approaches hinder concurrent access and parallelism.

SIAS employs the multi-versioning scheme to increase concurrent access and foster parallelism (Chapter 3)

- Invalidation Model: the versioning model of in-place invalidation creates small in-place (random) updates which are unsuitable for the properties of Flash storage.

  SIAS changes the *invalidation model* to facilitate out-of-place invalidation (Section 4.1).

- Version Organization Scheme: organization of tuple versions, addressing each of them as individual data items, incurs in-place updates of visibility meta-data.

  SIAS introduces a *new versioning scheme* that addresses all tuple versions of a data item as a unique set (Section 4.3).

- HDD Read Optimized Data Structures: read optimized data structures and optimizations such as clustering incur in-place updates and create mixed loads. Write optimized append storage is more suitable for asymmetric storage. The trend towards larger I/O transfer units (large page I/O) does not hold for Flash storage.

  SIAS reconsiders access methods to optimize for write access (Chapter 5). SIAS enables append storage management in *tuple granularity* (Chapter 6).

- Logical Page Append Storage Management: LbSMs that are implemented as a separate layer between the DBMS and the storage, operate in page granularity and create write amplification (smallest granularity to append is a page, even if just a single bit in the page is updated).

  SIAS significantly reduces the write amplification by integrating the LbSM into the MV-DBMS and appending in tuple granularity (Chapter 6).

- Free Space Management: maintaining free space per page is not feasible for LbSM on asymmetric storage. It inflates space consumption and avoids efficient cache usage.

  SIAS does not use free space reservation. SIAS employs write retention and allows for a *simplified buffer management* (Chapter 7).

- Traditional buffer managers optimize for hitrate: optimizing for locality and grouping of writes, forming small sequential writes (or appends) becomes a major point.

  The *Flash based adaptive replacement cache* (FBARC) has been developed (Chapter 7) to address the benefits of write clustering.

The combination of the presented approaches leads to:

- Reduction of write amplification: SIAS leads to a significant reduction of the write amplification that is incurred by the version organization and the invalidation scheme.

- Fewer host writes: SIAS issues fewer host writes. Tuple versions become the net amount of data pages.

- Lower space consumption: amount of occupied storage space is reduced since less net amount of data has to be written and no free space reservation is applied.

- Avoidance of in-place updates caused by the version organization and invalidation scheme.

- Flash awareness: complementing the FTL leads to more efficient and predictable garbage collection and wear leveling. Avoidance of logical to physical block-address (LBA to PBA) re-mappings that are caused by in-place storage management.
- Faster I/O of densely packed units: write less data and transfer more tuple versions (no free space reservation).
- Lower latency: I/O is completed faster, leading to latency reduction and increased transactional throughput on high loads
- Longevity: enhanced lifetime of Flash cells due to fewer host writes, avoidance of in-place updates.
- Performance: SIAS achieves higher transactional throughput and lower average response times.
- Shifting the knee-point: lower I/O latency and higher I/O throughput gracefully tolerates workload fluctuations and spikes.

1.10 Structure of the Dissertation

Chapter 2 provides an exposition of the properties of asymmetric storage technology such as Flash and Non Volatile Memory. This Section contains specifications provided by the manufacturers and our own experiments on the I/O behavior of Flash SSDs, where we built and configured a high end server system.

Chapter 3 discusses the transactional model, the principle of Multi Version Concurrency Control (MVCC) and distinguishes MV-DBMSs from temporal databases.

Chapter 4 comprises the version management of the MV-DBMS. The basic types of the invalidation model are considered first and the Snapshot Isolation algorithm is examined.

The concept of the SIAS approach is stated in Section 4.3, which is embodied in the *SIAS-Vectors* prototype (Section 4.4). With SIAS-Vectors we analyze the potential of the SIAS approach and develop the improved *SIAS-Chains* concept which is detailed in Section 4.5. SIAS-Chains is evaluated on different Flash storage, such as USB Flash, a single enterprise-class SSD and an SSD RAID compound. The evaluation and discussion of SIAS-Chains is contained in Section 4.5.5.

Section 4.7 contains the SIAS approach for simplified buffer management using write retention. The SIAS algorithm involves changes to the access methods. The revised methods for indexing and scanning are described in Chapter 5.

Different types and variants of LbSMs are examined in Chapter 6, focusing on append granularity and data placement. Optimizations to LbSMs are discussed in Section 6.5.

Section 7.2 describes the Flash-aware buffer management of FBARC. FBARC extends ARC by a write list utilizing the spatial locality of evicted pages to produce (semi-)sequential write patterns.

The related work is contained in Chapter 8 giving an overview of existing approaches and our own work and delivering a short survey of related approaches.

Chapter 9 contains a summary, a discussion of the contributions and future work.

2 New Storage Technologies

Contents

2.1 Flash Memory 21
2.2 Solid State Drives 24
2.3 Flash Endurance 28
2.4 Hardware Specifications 31
2.5 SSD RAID Configuration 34
2.5.1 Sylt Testbed 35

The properties of asymmetric storage technologies such as Flash differ from those of symmetric storage media such as HDDs. Most DBMSs were designed to compensate for the properties of HDDs, they treat SSDs as HDD replacements and are not optimized for their special properties. Properties of HDDs upon which MV-DBMSs were designed are well known:

- HDDs are symmetric storage and reads are equally fast as writes.
- They incur high latency since on an arbitrary random access (read or write) the head has to be positioned and the underlying disk has to rotate to the correct position.
- Physical limits are reached due to these mechanically moving parts.
- Over the past decades the access latency decreased only slightly, while data density, bandwidth and processing power have increased exponentially.

With asymmetric storage such as Flash and Non-Volatile Memories (NVMs), this fact is not true anymore. In the past years Flash based storage emerged as the main storage alternative to traditional HDDs. The key drivers of this development are the performance benefits of Flash-based storage and falling prices, such as price per GB and more importantly per IOPS. Capacities have significantly increased and modern physical interfaces and I/O protocols have been developed. In the following Section 2.1 the properties of Flash memory chips are introduced. SSDs are described in Section 2.2.

2.1 Flash Memory

Flash is persistent storage of a type of electrically erasable programmable read-only memory (EEPROM) which is comprised of either NAND or NOR Flash memory chips. NOR Flash memory is byte-addressable and mainly used for storing code. NAND chips are block-addressable and mainly used for storing data.

In the following we focus on NAND based Flash. Such Flash memories are available from manufacturers such as Hynix, Intel Micron Flash Technologies (IMFT), Powerchip, Samsung, Sandisk and Toshiba or different OEM resellers.

There are mainly three types of NAND storage available on the market:

- Single-level cell (SLC) NAND, capable of storing 1 bit per memory cell (bit-codes *0, 1*).

- Multi-level cell (MLC) NAND, capable of storing 2 bits per memory cell (bit-codes *00, 01, 10, 11*).

- Triple-level cell (TLC) NAND is capable of storing three bits per cell, also known as 3bit MLC (bit-codes *000, 001, 010, 011, 100, 101, 110, 111*).

Based on these types there exist further subtypes of NAND Flash chips. *Enterprise MLC (eMLC)* is a special type of MLC NAND which is designed for server storage that yields enhanced endurance. In comparison to consumer MLC it supports a higher amount of (over-) write cycles and therefore yields a higher durability: consumer MLC supports approx. 3000 to 10000 write cycles and eMLC 20000 to 30000 write cycles [25].

3D V-NAND comprises either MLC or TLC NAND, features higher capacity than two dimensional layouts of equivalent types, by deploying the cells in a three dimensional layout [98]. The "V" is an abbreviation for the vertical alignment of the NAND cells.

SLC NAND differs from MLC (and TLC) in lifetime, performance, density and costs. SLC yields lower response times and a 10 times longer lifetime than MLC. The performance and endurance of SLC comes with a price, it features lower density and involves higher cost. Therefore, it is mainly used in special storage environments. The majority of Flash based storage currently available on the marked is based on MLC or TLC Flash chips.

Some selected specifications are contained in Table 2.1. The Samsung 840 EVO series uses an optimization called *Turbo Write* which instruments the MLC NAND Flash in the same way as SLC chips, writing only one bit instead of three. This decreases the capacity but increases the performance of the device. The device internally runs a background maintenance process which migrates the data previously written in Turbo Write mode to the *physical* MLC format to reclaim unused capacity.

Table 2.1: Vendor specifications (excerpt): Intel X25-E 64GB [50], Samsung 840EVO [100]. WC = Write Cache.

Model	Intel X25-E 64GB	Samsung 840EVO
Application	Server, Data Center	Client PC
Type of Flash	SLC	3bit MLC (3D VNAND)
Model Number	SSDSA2SH064G1GC	MZ-7TE750, MZ7TE1T0
Queue Depth (QD)	Max. 32	Max. 32
Sequential Read	250MB/s	540MB/s
Sequential Write	170MB/s	520MB/s (Turbo Write)
Random Read 100% Span (QD 32)	35000 IOPS (8KB)	98000 IOPS (4KB)
Random Write 100% Span (QD 32, WC On)	3300 IOPS (8KB)	90000 IOPS (4KB)
Latency Read	75μs (QD 1)	unknown
Latency Write	85μs (QD 1, WC on)	unknown
Capacity	64GB	750GB, 1TB
Interface	SATA 3.0 3GB/s	SATA 6GB/s
Factoring Lithography	50nm	19nm
Endurance	2 PB random writes	3 years warranty
Trim support	No	Yes
On Device Cache	unknown	1GB DDR2 SDRAM

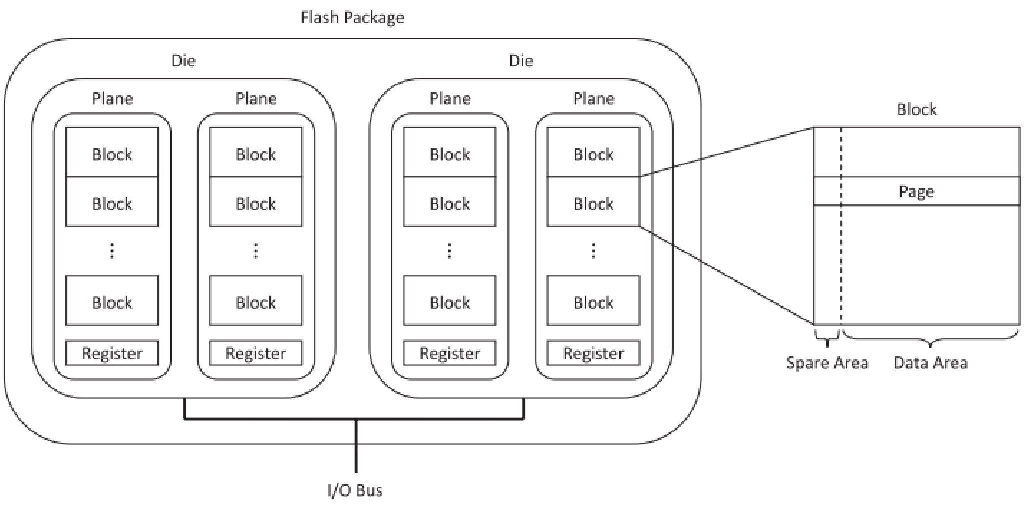


Figure 2.1: NAND Flash package layout [75]. Each Flash package comprises multiple dies which consist of several planes. A plane comprises blocks (erase-unit) which are divided into Flash-pages. Flash-pages are the unit of I/O and are accessed through the block device interface. Blocks are the unit of erase operations.

The common layout of a Flash memory package/chip comprises one or more *dies* which are individually segmented into multiple *planes*. Each plane contains several thousand *blocks*, each containing 64 to 128 Flash-pages that yield a data storage capacity of 2KB to 8KB and some space (e.g. 64-128 bytes) for storing meta-data such as Error Correcting Codes (ECC). The exact design (storage, capacity, amount of meta-data, etc...) of a Flash memory package varies from vendor to vendor. The Flash package layout is depicted in Figure 2.1 [75]. The write/read unit is a Flash-page (via the block device interface). It is technically impossible to overwrite Flash memories (only reset cells can be set), a write is always executed on a *clean/erased* block. In order to overwrite a block, it has to be deleted first. The erasure of a Flash-page is only possible when the whole block it is contained in, is erased first (bad block management). Hence an update operation issues a so called *erase-cycle* where the block is read into the cache, merged with the new data and written back after the physical block has been erased. Each erase-cycle increases the wear of the block, hence decreasing its lifetime. In the following such a block is referred to as an *erase unit*.

Most Flash chips provide a spare area per Flash-page which stores *out-of-band* (OOB) data, including error correcting codes (ECC), the state bits of the page and the logical page number (LPN) or the logical block number (LBN), respectively.

Technically, data is stored as a charge in a floating gate of the semiconductor within the Flash cells. For instance, MLC induces four different charge levels in order to store two bits of information per cell. This has direct influence on endurance, since with smaller (sub-micron) cell design sizes the amount of reliable set and retrieve operations also decreases. Hence it is critical to avoid write patterns that frequently update data on the same Flash cells. Flash memories exhibit the following important properties:

- Asymmetric read/write performance: Access latencies for read operations vary from nanoseconds to several microseconds [19]. Random read and sequential read performance (throughput) is roughly equally fast. Write operations, are at least an order of magnitude slower. Write latencies for random and sequential access vary significantly.

- Three operations: read, write and erase that need to operate in different granularities (pages, blocks).
- No mechanical latency.
- No overwrites: need for *erase-cycles* that wear out the Flash memory cells.
- Varying physical addresses: as a result of erase-before-overwrite and wear-leveling.
- Inherent parallelism - each die can operate independently.
- *Mixed load performance*: the mixed-load performance of Flash devices is sub-optimal compared to their *pure* read or write performance (Figure 2.8 and Figure 2.9).
- *Endurance*: writes to the same memory location can cause the memory cells to malfunction hence the notion of *endurance* or *longevity*.

2.2 Solid State Drives

In the past few years Flash based storage for DBMSs became a standard technology and the main alternative for HDD based storage systems. Solid State Drives (SSDs) are integrated devices which make Flash memory accessible, emulating an HDD and introduce additional properties. SSDs comprise Flash memory chips, a controller that runs software logic that manages the Flash chips, a volatile cache (SDRAM) and a physical interface to connect the device. Figure 2.2 depicts the general layout of a Flash SSD.

The *Flash Translation Layer* (FTL) subsumes the software logic that manages the Flash chips. For backwards compatibility and interchangeability SSDs implement legacy hardware interfaces, addressing and physical space management methods which were designed for properties and characteristics of legacy block-device hardware. They implement a block device interface in order to be compatible and interchangeable with traditional storage, such as Hard Disk Drives (HDDs). This fostered their proliferation, since traditional DBMSs can operate on them. It enables *simple replacement* and delivers an immediate performance boost. This fast *"sugar-Flash"* comes at a price. SSDs are prevented from realizing their full potential. Even worse, access methods that were developed for legacy storage can be harmful for the Flash memories.

Flash SSDs are available for different storage systems, yielding different block-device interfaces, connectors and form factors. Serial ATA (SATA) is commonly used for consumer PCs and smaller (inexpensive) server systems. Serial Attached SCSI (SAS) and PCI Express (PCIe) is used for server storage and reaches approx. ten times the performance of consumer models. Parallel ATA and SCSI are successively replaced by SATA and SAS. SSDs for mobile devices mainly implement a M.2 or mSATA interface.

In the following the most relevant properties and characteristics of Flash Solid State Drives (SSDs) are listed.

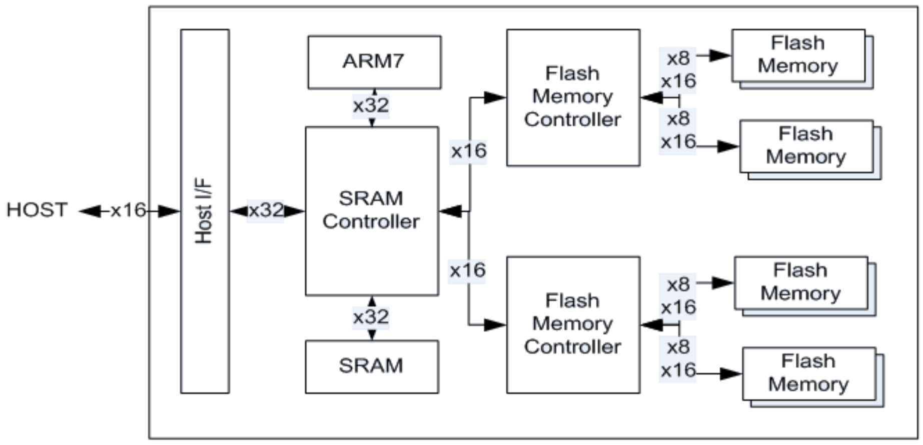


Figure 2.2: General layout of a Solid State Drive (SSD). A general SSD is comprised of several Flash memory chips that are connected to memory controllers. A SRAM controller (ARM) manages the connection to the host interface. The FTL subsumes SRAM and Flash controllers, ARM processor and volatile SRAM cache.

Flash Translation Layer

The on-device Flash Translation Layer (FTL) of Flash storage media provides backwards-compatibility, legacy interfaces (block device) and hides Flash storage specific behavior. The essential mechanism that is performed by the FTL is the *address translation*. The principle of the address mapping is depicted in Figure 2.3. The address mapping, along with the translation scheme, is implemented within the Flash SSD as a black box and invisible for accessing layers and several maintenance procedures are entailed with the address translation. When a page is requested by the application, the logical block address (LBA) is directed through the block device interface to the SSD's FTL. The FTL performs a lookup within the address mapping table, resident in the SRAM of the SSD, to determine the physical block address (PBA). Subsequently the physical page is returned.

Since new data (updates, inserts) always has to be stored on clean (newly allocated) pages, the physical address of already written pages is prone to change. The logical address, visible outside the black-box, stays the same for the application (e.g. the database). The physical address is not visible to the application.

This leads to unpredictable behavior, since several FTL-background processes such as garbage collection, metadata synchronization, wear leveling and eventual log-block merges are executed in the background and cannot be instrumented. For instance, the address translation process may invoke the garbage collection once the amount of physical free space drops below a certain threshold.

There exist mainly three different mapping schemes:

- *Block-level mapping*. Block-level mapping operates on Flash-block granularity. Hence it abstracts from Flash-pages. The advantage of block-level mapping is the lower SRAM overhead. The main disadvantage of block-level mapping schemes is that due the higher granularity of the mapping it is not efficiently possible to seperate hot from cold Flash-pages. Hot Flash-pages are updated frequently, while cold pages stay unaltered. This creates a high amount of invalid Flash-pages. In order to reclaim space the garbage collection mechanism has to copy (migrate) valid pages to a new Flash-block before the old

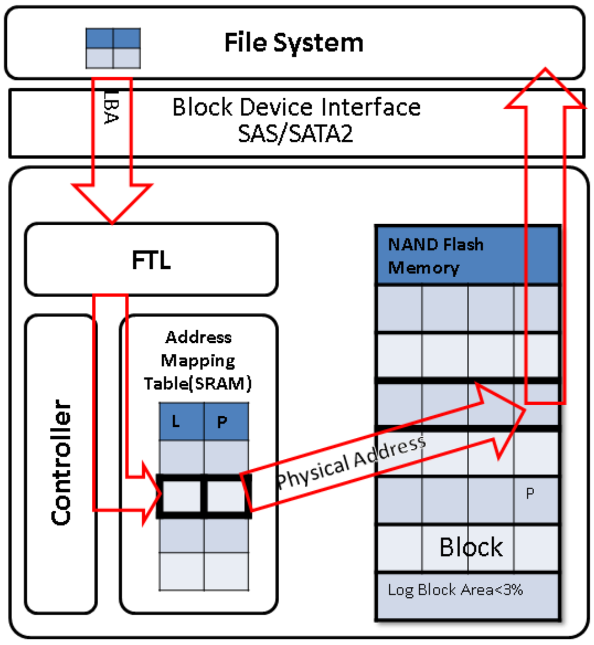


Figure 2.3: Flash Translation Layer Mapping. A page is requested by the application using the logical block address (LBA). Through the block device interface the request is directed to the FTL which performs a lookup in the address translation table to retrieve the physical address.

block is erased. A mixture of hot and cold pages therefore degrades the performance of the garbage collection ([75])

- *Page-level mapping*. Page-level mapping schemes store a mapping of each logical block address to a physical block address. This incurs a high overhead on the SRAM cache and increases the size of the mapping table. Scalability is an issue and growing capacities of Flash SSDs render pure page-level mappings infeasible. Page level mapping is mainly used on devices with smaller capacities and NOR based Flash storage.

- *Hybrid mapping*. Hybrid mapping schemes take advantage of the flexibility of a page-level mapping while keeping the SRAM overhead reasonably low, close to that of block-level mapping. One main task of this approach is to identify and manage hot and cold pages.

A summary of FTL mapping scheme approaches is given in the related work in Section 8. A thorough survey of different FTL mapping schemes is also provided in [75].

Asymmetric Storage

Reads are up to an order of magnitude faster than writes (measured in I/O operations per second – IOPS). This property is inherited by the Flash memories. On Flash SSDs the I/O asymmetry is particularly large in the case of random writes [18]. These are one to two orders of magnitude slower than random and sequential reads respectively [18]. Some manufacturers of SSDs report *symmetric* read/write performance for their products, yet this is only true under certain circumstances and an effect of the on-drive volatile cache. As soon as the cache is filled, the asymmetric property of the Flash memories is revealed. Write as well as read performance drops after that point. The cache quickly gets filled during a continuous operation, hence this property can't be masked. The drop in read performance is also caused by the filled cache, since mechanisms such as *read ahead* become inefficient. Random (re-)writes have long-term effects on performance and endurance.

Costly Erase Cycles

Overwriting a physical block on Flash directly is impossible. Either a read-update-delete operation (erase-cycle) has to be performed or the block has to be remapped. On the event of a remapping operation the FTL writes the updated block to a different physical location and stores the mapping data. Since the erasure of a block is only possible after the execution of an erase-cycle (erase unit is read into the cache, merged with the new data and written back after the physical erase unit has been erased). Since the read-update-delete operation is costly, most Flash SSDs postpone the erase operation to a later time. They use the on-drive cache and a predefined spare area (over-provisioning).

Performance

The random read performance is exceptionally good (latency and throughput), especially on small block sizes (4KB/8KB [91]). The access time is bound by the transfer time and not the seek time any more. The larger the blocksize, the higher the transfer time, resulting in fewer IOPS. Figure 2.4 depicts the throughput of the SSD and HDD, measured in IOPS, on different blocksizes. This contradicts the present trend towards larger database page sizes for HDDs, since the cost of fetching a small page is low and can be parallelized in SSDs.

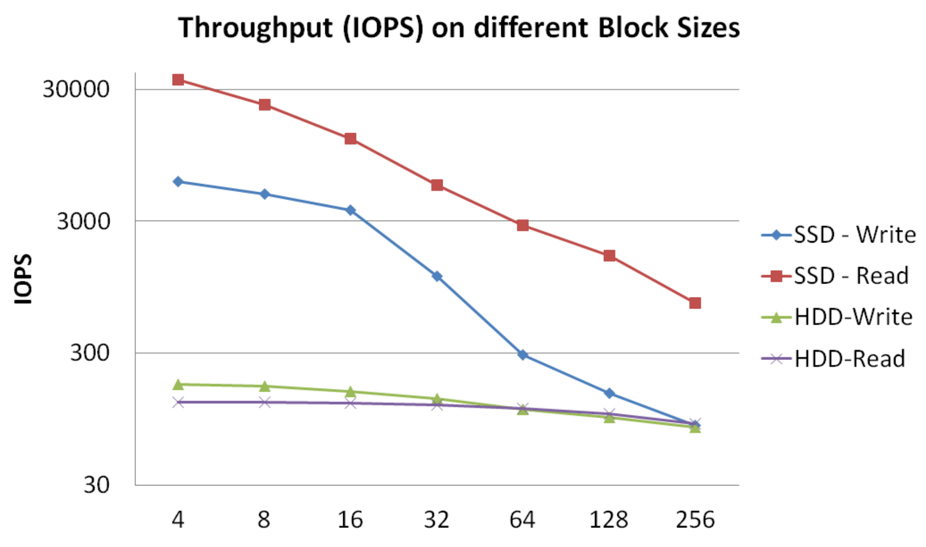


Figure 2.4: SSD and HDD throughput on different block sizes. Random read and random write. IOPS scale is logarithmic.

Random write performance is lower than the random read performance (IOPS). Especially on in-place updates the random write performance is significantly lower than the random read performance [18]). Nonetheless, the random write throughput is an order of magnitude better than that of an HDD. (Small) random writes, especially updates, are an issue not only in terms of performance and durability of the Flash cells, they lead to long-term performance degradation due to Flash-internal fragmentation. Some Flash device manufacturers 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. The performance of the Flash device is bound by the characteristic performance of the Flash memory, yielding unstable performance. As soon as the cache is filled the Flash-specific characteristics are revealed.

Sequential or semi-sequential write patterns are preferred in terms of performance. When a set of random write I/O requests is concentrated on a relatively small region, they are performed nearly as fast as sequential requests (Section 7.2, [29, 13]). Append Log Storage is another approach that sequentializes write requests and is studied in Chapter 6.

Sequential operations are also asymmetric. Sequential reads are approx. 25% faster than sequential writes and sequential writes are significantly faster than random writes. These figures are reduced to 25% due to caching and read ahead mechanisms.

Mixed workloads are slower than segmented loads. Due to the layout of the Flash memories a mixture of read and write operations as well as random and sequential access patterns performs slower than a *pure* work-load.

I/O Parallelism

Individual Flash chips can handle multiple I/O commands simultaneously. This is due to their internal organization. Each Flash-chip comprises several dies, where each die can operate on its own, independently of other dies, and is capable to execute multiple different commands in parallel. Figure 2.1 depicts the internal layout of a single Flash package/chip. On a Flash device there exist multiple levels of parallelism, since a Flash device is comprised of multiple Flash chips. In addition Flash devices comprise multiple independently connected Flash chips in an array-like organization. Compared to the typical hard drive a Flash device can handle much higher levels of I/O parallelism. Furthermore, Flash devices (SSDs) can execute much more I/Os in parallel than the SATA interface allows. The I/O performance increases with the number of asynchronous I/O requests (queue depth) [3, 18].

Wear

Flash memories are prone to wear and offer limited endurance and durability. Each flash chip is divided into erase-units (Flash-blocks) that can only be written a finite number of times (erase-cycle), delivering increasingly unreliable results. Wear leveling (WL) is executed within the FTL which balances the write load to evenly wear out the Flash memory chips. WL distributes the amount of overwrites (erase cycles) evenly across the Flash cells. When Flash cells wear out the whole block in which they are contained is marked as defective and the reserved over-provisioning area is used. The block is physically relocated and its valid contents are written to different block on the Flash device. Relocations are stored in the FTL's mapping table. Over-provisioning is mainly used to postpone expensive erase operations that are triggered by updates. Instead of updating a block in-place the block is remapped to a different location. The efficiency can be influenced by the user of the SSD, by increasing the amount of unused space.

2.3 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

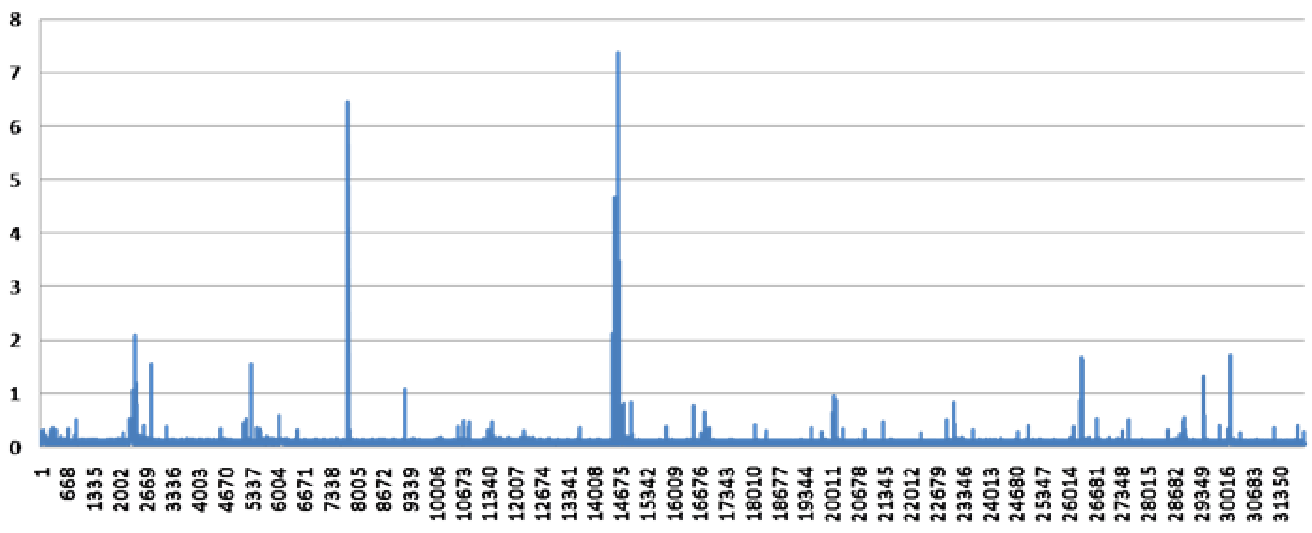


Figure 2.5: Latency outliers on random write load on a single SSD with factory clean state. Duration 60 seconds and 335k write requests. The abscissa is showing the number (in order) of each write request. The ordinate is showing the latency of each request.

SIAS algorithm significantly reduces the write amount and the required space on the device. In-place updates are avoided, new pages are written (appended) in monotonically increasing order.

Masked Behavior

The FTL emulates an HDD compatible block device interface and hides internal background processes that manage the Flash memory. A black-box abstraction is created that hides internals and specifics of the Flash memory. The basic background processes are represented by a logic block mapping, the garbage collection and wear leveling algorithms. These background processes run asynchronously and are triggered on certain conditions based on the device specific state.

This organization leads to a crucial property: *unpredictable performance* in terms of latency and throughput. The black-box abstraction leads to an interference between the execution of foreground I/O requests (issued by the DBMS) and the background processes on the device.

When the (volatile) cache is filled during normal database execution, the performance drops since write operations have to be carried out. In modern SSDs there exists a so called *spare area* which is additional space available only for the internal device. Error correcting codes are also stored within that area. If the spare area is saturated or a block has been remapped again, expensive erase-modify-write operations have to be executed. Such executions produce *outliers* in latency and throughput, which is unpredictable to the software behind the device interface. Although the DBMS issues a *sequential* scan over its logical block addresses the SSD may have to find remapped physical blocks at different locations.

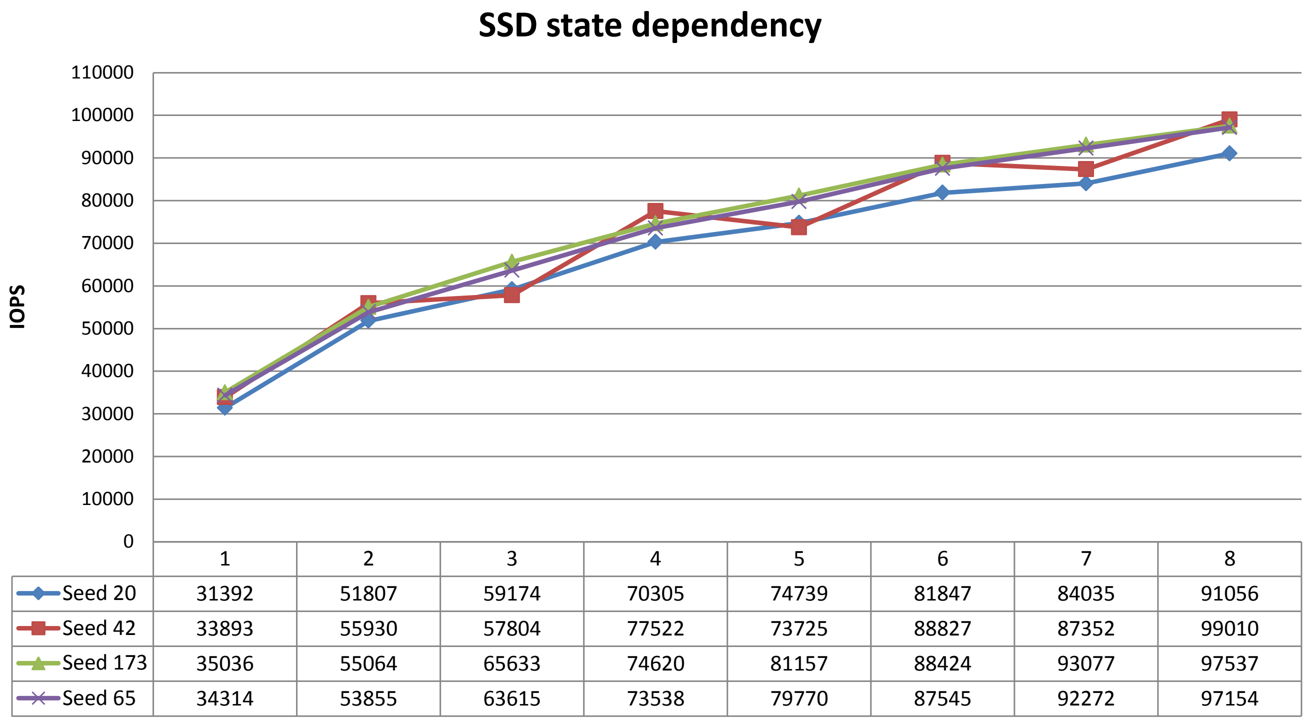
Unpredictable Performance.

The address mapping (Figure 2.3) and background processes that manage the Flash chips (garbage collection, wear leveling, etc.) and provide the *legacy* interface create a *black box abstraction* which creates state dependencies and prevents stable performance characteristics. Sporadic non-deterministic and workload dependent outliers occur, as depicted in Figure 2.5.

A single SSD is instrumented with 60 seconds of random write requests, with a total of 335k requests. We measured each write request's completion time. The difference between the average and maximum request latency is significant. The main reason for this behavior is *Garbage Collection*. The FTL runs a garbage collection (GC) process that is executed asynchronously. Used blocks are collected and co-located such that erase units can be erased to reclaim space. In most SSDs the GC process is triggered in the background when the device is idle or when a threshold (used up space) is reached.

State Dependency

Figure 2.6 shows the state dependency of the Intel X25-E SLC SSD. We conducted I/O benchmarks where each I/O trace is comprised of mixed random read and random write requests and is created with an individual seed. Each trace was run consecutively for eight times. After each individual test-run, the system was powered down and restarted in order to remove caching effects. With each consecutive run, using the same seed, the SSD performed better. After switching to a different seed, practically creating a different trace, the performance broke down to the point of each initial benchmark series. This emphasizes the existence of state dependency on Flash devices that are caused by internal processes and the occurrence of remapping on the device. Therefore, it is difficult to guarantee certain performance figures as long as the I/O pattern is not optimized.



	1	2	3	4	5	6	7	8
Seed 20	31392	51807	59174	70305	74739	81847	84035	91056
Seed 42	33893	55930	57804	77522	73725	88827	87352	99010
Seed 173	35036	55064	65633	74620	81157	88424	93077	97537
Seed 65	34314	53855	63615	73538	79770	87545	92272	97154

Figure 2.6: SSD state dependency. Each seed forms a series of 8 test-runs that are consecutively executed. After each run the system was powered off. When the same trace is executed, the SSD performs better - even after a power loss. Thus making it difficult to guarantee reliable performance figures.

Trim

If the host system is not aware of the underlying SSD storage it does not inform the SSD when a block is deleted, since usually only an entry within the filesystem is updated. Traditional HDD storage devices do not need the trim command since they are capable to overwrite blocks. Since the block has to be explicitly erased on the SSD, the SSD is not able to garbage collect that block if only the filesystem is updated. Therefore, the *Trim* command was introduced that informs the SSD about the deletion of blocks. This is beneficial for the efficiency of the SSDs background processes such as garbage collection and wear leveling. Trim is implemented in modern operating systems such as Windows 7, Linux (kernel 2.6.33) and Max OS X (10.6.8).

Flash Write Amplification

The general term of *write amplification* is the difference of *new data* that has to be written and the actual amount of data that has to be written. For example overhead that is generated by meta-data and (hardware/software) interfaces.

Flash write amplification (F-WA) is the result of data written by the SSD's controller divided by write operations issued by the host The host accesses the SSD through a block-device interface. If the host is unaware of the underlying Flash, it may issue overwrites that on the SSD trigger a costly erase-cycle - this has to be avoided in order maintain sustained performance, reliability and longevity. Operations such as the remapping of Flash-blocks or migrating of valid Flash-pages within a Flash-block, creates the F-WA. The numerical value of F-WA is calculated dividing the amount of data written by the SSD's controller by the amount of data that is written by the host. In the ideal case the number is smaller or equal to one. If it is smaller than one the data written by the host is compressed by the SSD's controller.

2.4 Hardware Specifications

The specification of our Intel X25-E SLC SSD are contained in Table 2.1. Table 2.2 and Table 2.3 show the measured performance of the Intel X25-E SLC SSD. The performance figures in Table 2.2 are recorded on a Windows Server 2008, while those in Table 2.3 are recorded on an Ubuntu Linux Server. The performance figures confirm the vendor's specification. Experiments on the device serve as an example of the basic properties of the Flash SSD. [99] provides further analysis of different Flash devices.

Table 2.2: Single Intel X25-E SSD performance figures under Windows 2008 Server. With and without file system. Random request size 4KB. Sequential request size 512KB.

Property	IOMeter RAW	IOMeter NTFS	XDD NTFS
Random Read (IOPS)	35661	35937	34667
Random Write (IOPS)	5179	5893	5732
Sequential Read (MB/s)	255	261	265
Sequential Write (MB/s)	195	194	195

Mixed Workloads

As already stated, mixed workloads are slower than segmented loads. Figure 2.8 depicts the mixed workload performance of a single SSD (Intel X25-E). The tests were conducted using

Table 2.3: Single Intel X25-E SSD performance figures under Ubuntu Linux Server. With and without file system. Random request size 4KB. Sequential request size 512KB.

Property	IOMeter RAW	IOMeter XFS	XDD RAW	XDD XFS
Random Read (IOPS)	35700	35792	35941	35068
Random Write (IOPS)	4923	4602	5659	5623
Sequential Read (MB/s)	248	257	252	248
Sequential Write (MB/s)	197	196	197	196

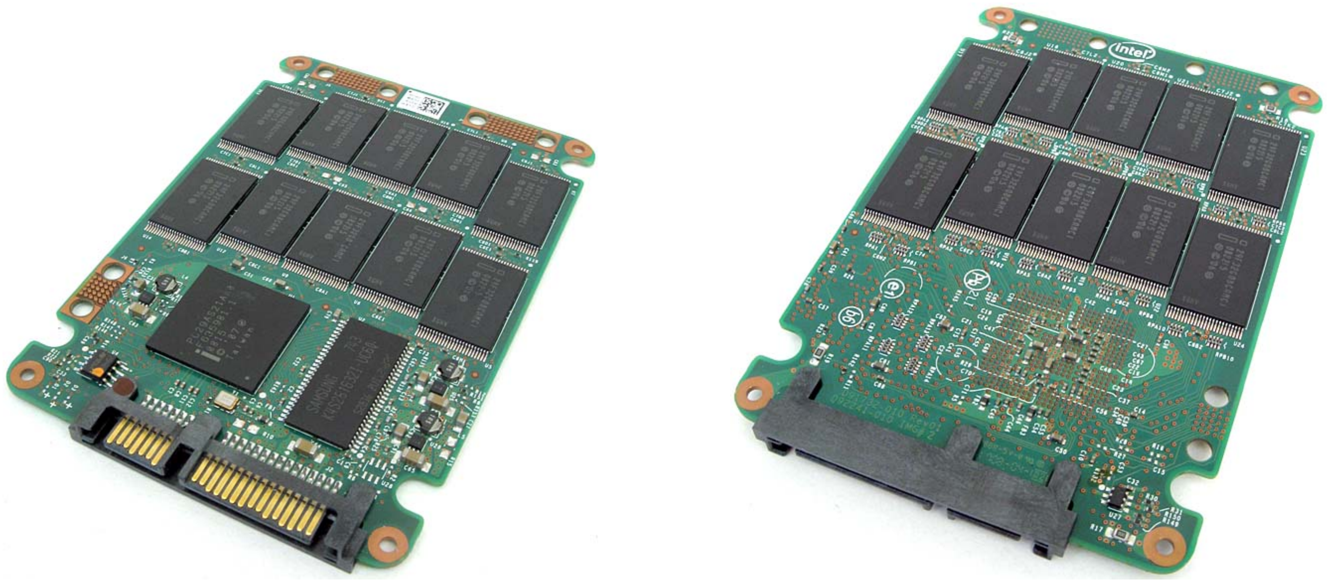
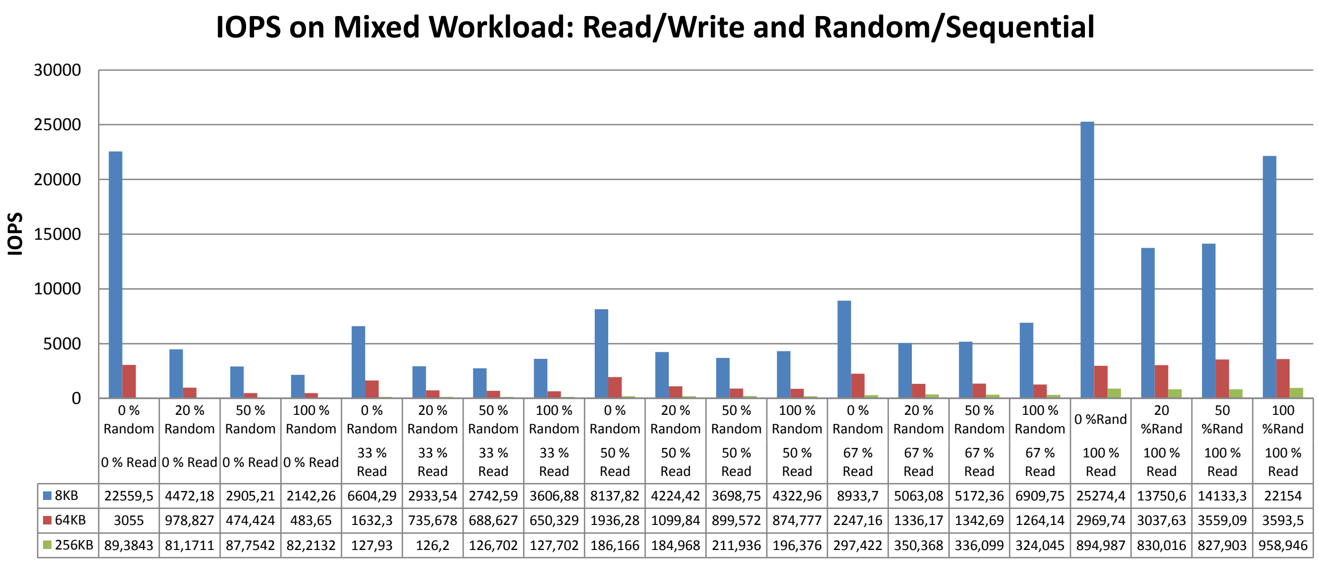


Figure 2.7: Intel X25-E SSD without the protective case. The device provides an SATA interface and power connector. It comprises 16 SLC-NAND Flash chips, the controller and volatile cache (SD-RAM). Source Intel [50]

the IOMeter benchmark tool with activated write cache and read ahead on the SSD. Activated write cache and read ahead significantly improves the performance of full sequential writes (no mixture with reads) and full sequential reads (no mixture with writes), respectively. Write performance decreases with increased randomness. Figure 2.8 shows a significant drop in write IOPS when only 20% of the writes are random writes. When full sequential write access is mixed with only 33% sequential read access the performance drops by approximately 70% (8KB block size).

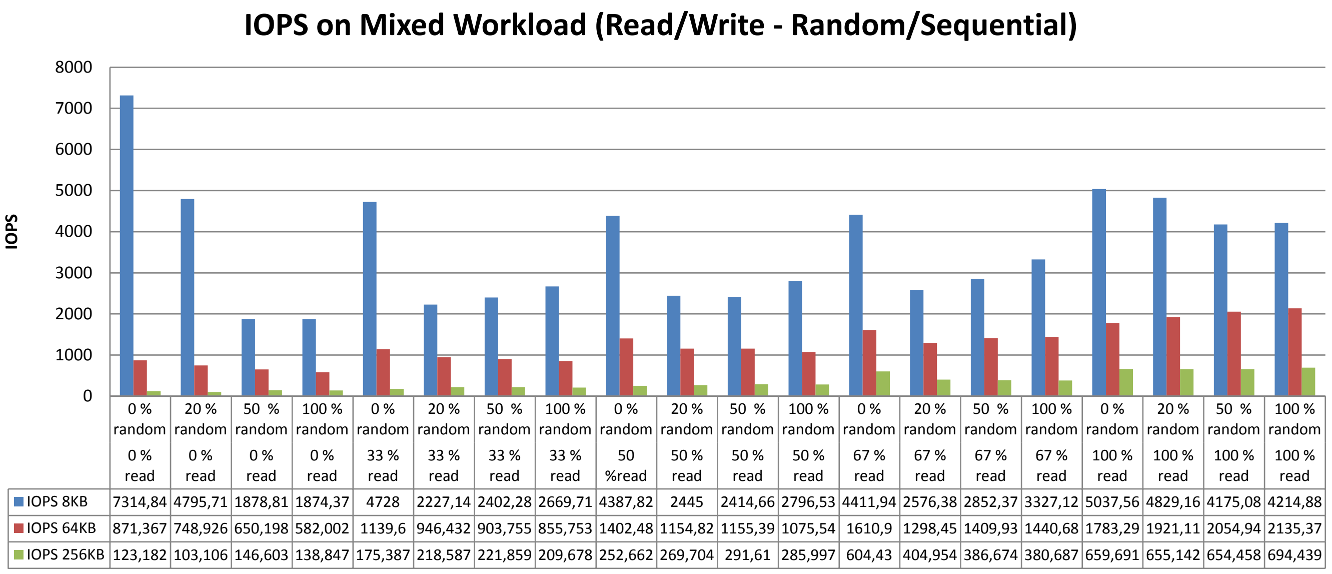
In general, random reads are less costly than random writes and small random writes are especially expensive, whereas sequential writes are less costly than random writes. When introducing randomness the read ahead has to "guess" the correct read pattern, which leads to an initial decrease. Read IOPS increase on further increment of the randomness.

In contrast Figure 2.8 depicts the effect of deactivated write cache and no read ahead. The overall write performance is impaired the most on small block sizes, since each issued I/O has to be directly executed on the device - it can't be cached. Small random writes are slower than sequential or semi-sequential writes. Larger block sizes (256KB) are less impaired on sequential access. It is obvious that small random writes benefit the most from activated write cache. Overall writing in large and sequential access patterns and reading in smaller units is beneficial



	0 % Random 0 % Read	20 % Random 0 % Read	50 % Random 0 % Read	100 % Random 0 % Read	0 % Random 33 % Read	20 % Random 33 % Read	50 % Random 33 % Read	100 % Random 33 % Read	0 % Random 50 % Read	20 % Random 50 % Read	50 % Random 50 % Read	100 % Random 50 % Read	0 % Random 67 % Read	20 % Random 67 % Read	50 % Random 67 % Read	100 % Random 67 % Read	0 %Rand 100 % Read	20 %Rand 100 % Read	50 %Rand 100 % Read	100 %Rand 100 % Read
🟦 8KB	22559,5	4472,18	2905,21	2142,26	6604,29	2933,54	2742,59	3606,88	8137,82	4224,42	3698,75	4322,96	8933,7	5063,08	5172,36	6909,75	25274,4	13750,6	14133,3	22154
🟥 64KB	3055	978,827	474,424	483,65	1632,3	735,678	688,627	650,329	1936,28	1099,84	899,572	874,777	2247,16	1336,17	1342,69	1264,14	2969,74	3037,63	3559,09	3593,5
🟩 256KB	89,3843	81,1711	87,7542	82,2132	127,93	126,2	126,702	127,702	186,166	184,968	211,936	196,376	297,422	350,368	336,099	324,045	894,987	830,016	827,903	958,946

Figure 2.8: Variations of workload types on a single Intel X25-E SSD with activated write cache and read ahead. Mixed workloads (read and write or sequential and random) are slower than segregated loads. Sequential writes are less costly than random writes.



	0 % random 0 % read	20 % random 0 % read	50 % random 0 % read	100 % random 0 % read	0 % random 33 % read	20 % random 33 % read	50 % random 33 % read	100 % random 33 % read	0 % random 50 %read	20 % random 50 % read	50 % random 50 % read	100 % random 50 % read	0 % random 67 % read	20 % random 67 % read	50 % random 67 % read	100 % random 67 % read	0 % random 100 % read	20 % random 100 % read	50 % random 100 % read	100 % random 100 % read
🟦 IOPS 8KB	7314,84	4795,71	1878,81	1874,37	4728	2227,14	2402,28	2669,71	4387,82	2445	2414,66	2796,53	4411,94	2576,38	2852,37	3327,12	5037,56	4829,16	4175,08	4214,88
🟥 IOPS 64KB	871,367	748,926	650,198	582,002	1139,6	946,432	903,755	855,753	1402,48	1154,82	1155,39	1075,54	1610,9	1298,45	1409,93	1440,68	1783,29	1921,11	2054,94	2135,37
🟩 IOPS 256KB	123,182	103,106	146,603	138,847	175,387	218,587	221,859	209,678	252,662	269,704	291,61	285,997	604,43	404,954	386,674	380,687	659,691	655,142	654,458	694,439

Figure 2.9: Variations of workload types on a single Intel X25-E SSD with deactivated write cache and no read ahead. Mixed workloads (read and write or sequential and random) are slower than segregated loads. Sequential writes are faster than random writes.

for SSDs. Write caching and read ahead mechanisms are limited on continuous workloads (24/7 OLTP). *In the ideal case all writes are issued sequentially in large units and reads are issued in small block sizes.*

2.5 SSD RAID Configuration

Building a large, Flash-based storage system that is capable of delivering the aggregated performance of each connected Flash device is a non-trivial task. Although recent SSDs reach capacities of up to 6TB, they have a higher price/tb value than the equivalent amount of SSDs that yield smaller capacities. The aggregated performance of an SSD RAID system largely depends on the layout and infrastructure of the whole system, e.g. whether hardware RAID controllers or software RAID is used.

Traditional hardware based RAID technology yields performance and scalability bottlenecks when set up with Flash SSDs [88]. Such configurations cannot handle more than a few SSDs until they get saturated. We examined different configurations of hardware and software RAID with the purpose to reach the peak performance of each configuration. Our research indicates that less than 10% of the expected random read rate and less than 35% of the expected sequential read rate can be achieved by a traditional hardware RAID configuration [88], [89].

RAID layouts amplify the properties of Flash SSDs, especially their read/write asymmetry, and write behavior. Hardware RAID controllers that are designed for the properties of HDDs are saturated by only a few SSDs and become a bottleneck. The distribution of data by the hardware controller amplifies fragmentation and write issues of the Flash SSDs, creating a negative impact on the overall performance.

With software RAID and host based storage the aggregated and theoretically possible peak performance can be reached.

Being capable of reaching peak performance by an adequate layout/configuration of the RAID storage subsystem is a necessity for the development of a DBMS architecture that is capable of effectively utilizing the performance. It is essential to ensure that no bottlenecks are artificially created by a suboptimal configuration of the hardware system.

In order to benchmark the I/O subsystem and layout of the hardware system, synthetic benchmarks are employed that are capable of selectively executing certain I/O patterns. This delivers the storage systems raw performance and proves that the configuration is capable of reaching the peak performance as the sum of its components.

With our *Sylt* server testbed we analyzed various configuration variants in order to determine the optimal hardware and software layout that is capable of reaching the maximum theoretical performance of the hardware. The final layout is depicted in Figure 2.11 and explained in the following Section.

2.5.1 Sylt Testbed

Figure 2.10: The Sylt Server testbed installed in the server rack.

We evaluated the layout and configuration of our Sylt server testbed (Figure 2.10) using various performance benchmarks such as FIO, XDD, IOMeter and DBT2. The specification of the Sylt Server is contained in Table 2.4. The baseline performance figures are contained in Table 2.5.

Table 2.4: Sylt specification.

Mainboard	Supermicro with QPI bus-system
Main memory	48GB
Storage Controller	Two LSI 9211-4i PCIe x4 - (host based storage)
SSD Storage	8x Intel X25-E SLC 64GB
Raid	Linux Software Raid, managed using mdadm
I/O scheduling	NO-OP scheduling
Main Operating System	Linux 2.6.34-gentoo-r12
Secondary Operating System	Windows 2008 Server

Figure 2.12 shows the random read performance of the Sylt server on raw storage. Incrementing the queue depth leads to higher IOPS for all blocksizes. The smaller the blocksize the higher the IOPS. Figure 2.13 shows the average read latency of random read access on different block sizes with incremental queue depth. Smaller blocksizes achieve lower response times on higher queue depth. Peak performance is only reached when the Flash SSDs can be saturated with requests. This occurs when the queue depth amounts to 256 (32 per SSD). Analogously Figure 2.14 shows the throughput of random writes and Figure 2.15 shows the latency in milliseconds. The same trend can be observed. This confirms the trend that transfer time becomes more important than seek time.

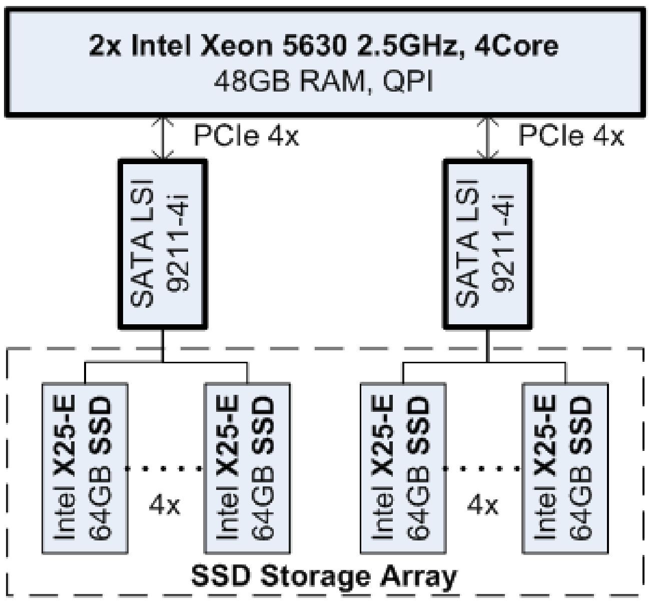


Figure 2.11: Final configuration of the Sylt Server testbed.

Table 2.5: Sylt baseline performance under Linux.

Property	Block Size	Read	Write
Sequential Throughput (MB/s)	256KB	2018	1477
Sequential Throughput (MB/s)	512KB	2027	1487
Sequential Latency (ms)	256KB	0.382	0.521
Sequential Latency (ms)	512KB	0.432	0.703
Random Throughput (IOPS)	4KB	279 776	41 882
Random Throughput (IOPS)	8KB	184 469	30 458
Random Latency (ms)	4KB	0.251	0.085
Random Latency (ms)	8KB	0.298	0.221

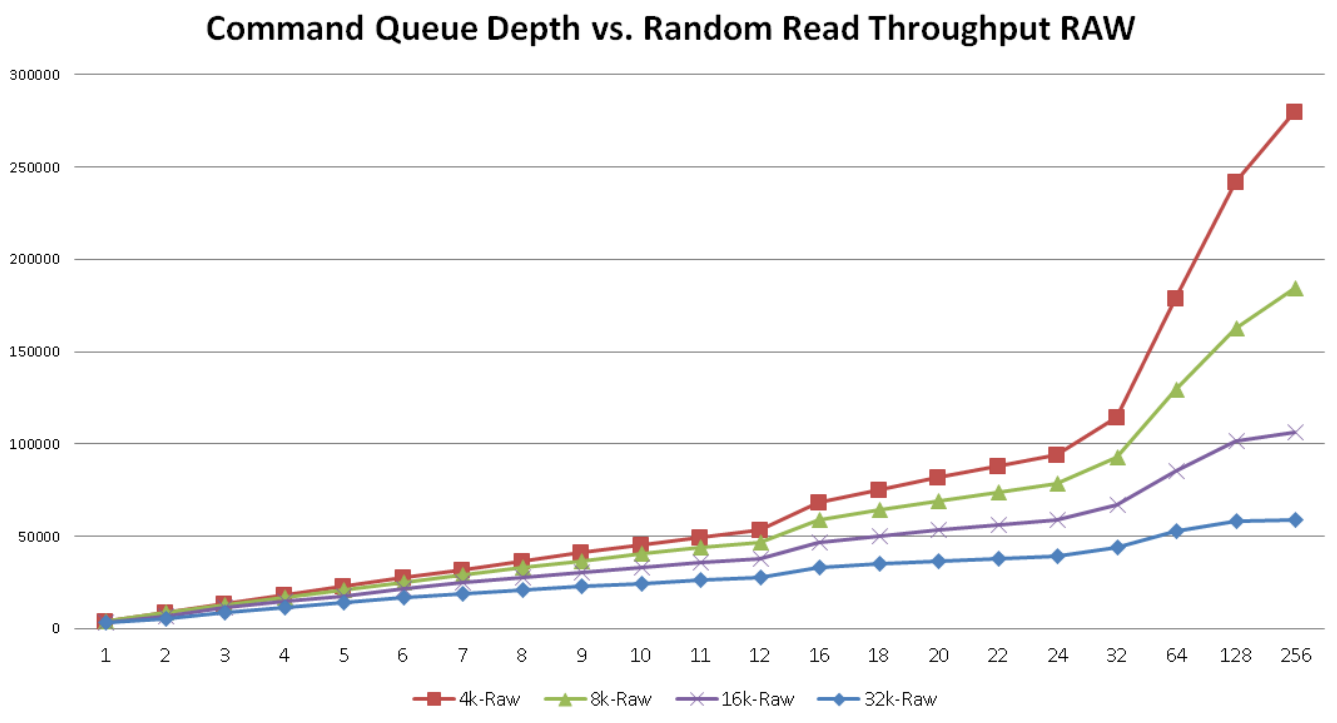


Figure 2.12: Sylt read IOPS on different block sizes and queue depths. Abscissa indicates the queue depth. Ordinate indicates throughput in IOPS. Each graph represents a different blocksize.

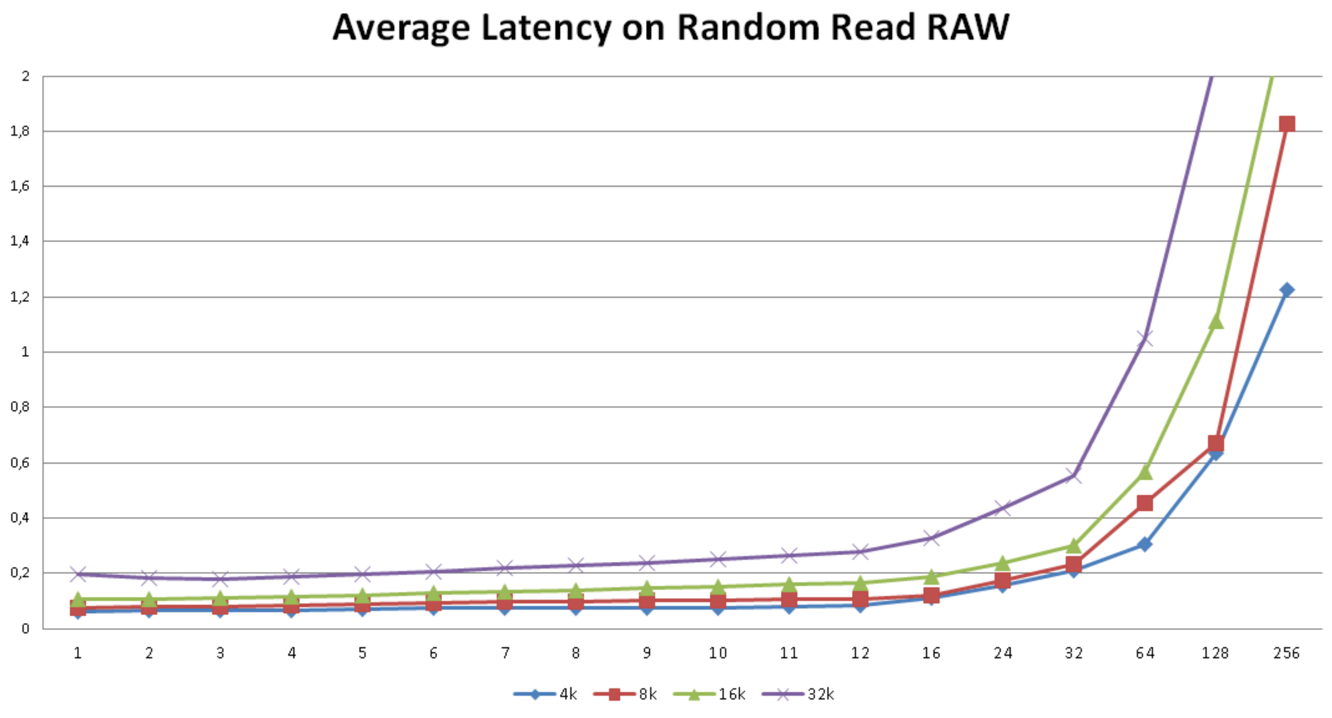


Figure 2.13: Sylt read latency on different block sizes and queue depths. Abscissa indicates the queue depth. Ordinate indicates latency in milliseconds. Each graph represents a different blocksize.

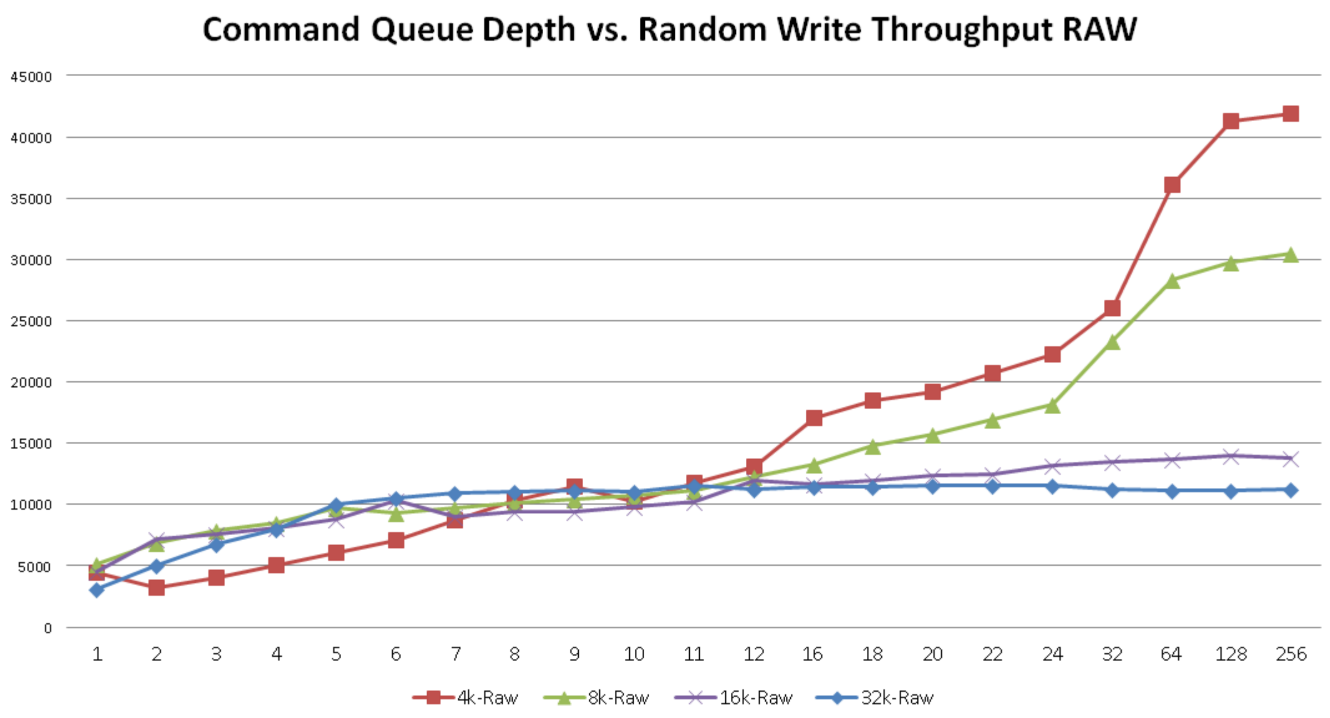


Figure 2.14: Sylt write IOPS on different block sizes and queue depths. Abscissa indicates the queue depth. Ordinate indicates throughput in IOPS. Each graph represents a different blocksize.

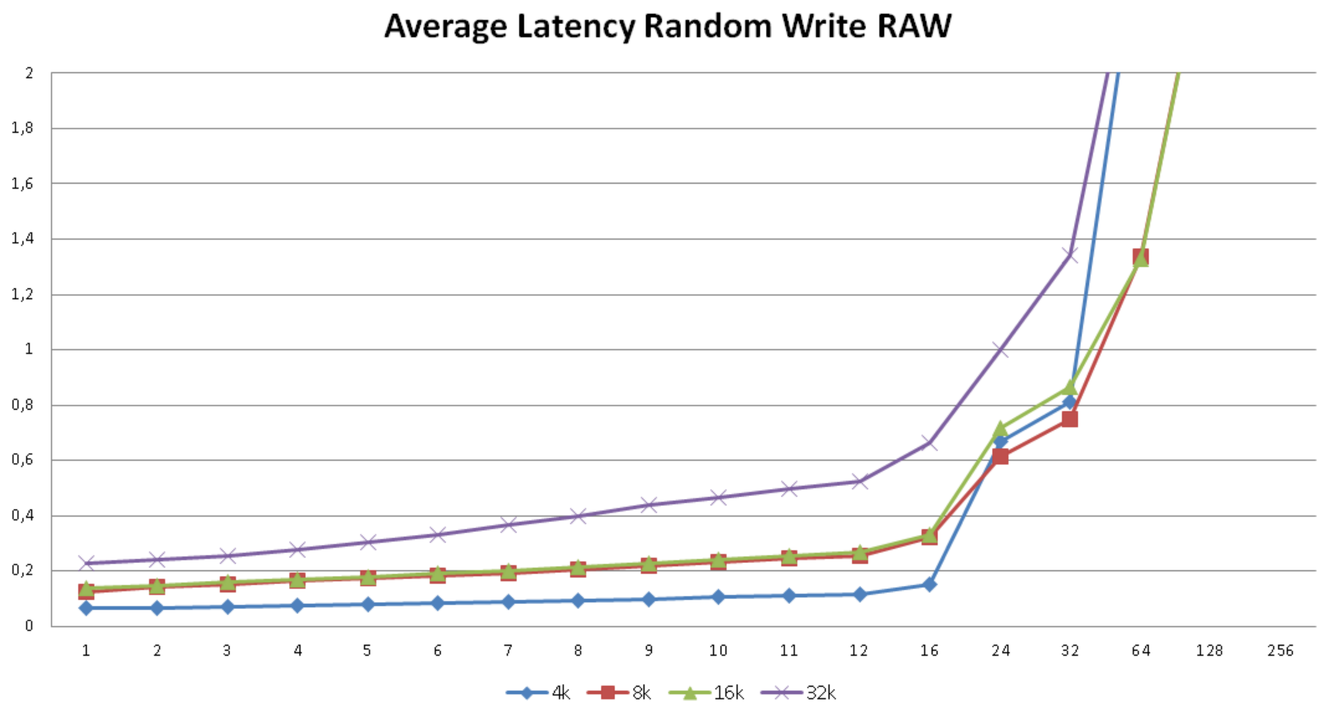


Figure 2.15: Sylt write latency on different block sizes and queue depths. Abscissa indicates the queue depth. Ordinate indicates latency in milliseconds. Each graph represents a different blocksize.

3 Transaction Management

Contents

3.1 Multi Version Concurrency Control 39
3.2 MV-DBMS 40
3.3 Multi Version Concurrency Control on SSD 40

Concurrent execution of transactions on the same DB requires a concurrency control mechanism. Uncontrolled execution of concurrent transactions that operate on the same data item(s) may produce an inconsistent state of the DB and anomalies may occur. To maintain a consistent state, it has to be ensured that the concurrent (interleaved) execution of multiple transactions maintains the ACID properties of atomicity, consistency, isolation and durability. In short the transaction manager has to guarantee that concurrently running transactions do not interfere with each other.

The transaction manager observes and controls the execution of transactions, where each transaction is described as a logical unit of work in terms of database processing. Transactions transfer the database from a consistent state to another consistent state.

Transaction management also includes the task of recovery control. The proposed architectural changes and algorithms in this work use existing recovery control mechanisms, such as write ahead logging (WAL). The database is separated from the (recovery-) log. Approaches to recovery control algorithms are not further investigated in this work.

3.1 Multi Version Concurrency Control

Management of multiple concurrent transactions is provided by the multi-version concurrency control mechanism (MVCC).

There exist different approaches to MVCC [105] such as *Timestamp Ordering*, *Multi-version Two-phase Locking* and *Snapshot Isolation*. Snapshot Isolation is a particular type of concurrency control scheme and further described in Section 4.2.

Multi-version Concurrency Control (MVCC) schemes create versions of data items once they are modified. On each write of a data item X a new version X_i is created. On a read request the appropriate version of X is selected by the concurrency-control manager. One major advantage of MVCC is the support of lock-free reading access. A read access is never blocked or delayed. Any number of read transactions and one write transaction can operate on the same data item. In a typical enterprise OLTP workload reading access (small read access) dominates and a certain amount of small, selective updates is issued. Individual transactions are *short* running and last only milliseconds. With respect to the OLTP workload and Flash storage, MVCC seems to be a good match. MVCC is also valid for workloads comprised of a mixture of long and short running transactions. Where *long* running transactions are characterized by more complex transactions that last minutes or up to several hours. The lock-free read access enables a higher degree of concurrency compared to classical two-phase-commit locking protocols that depend on (shared) read locks which cause delays by waiting for lock releases.

3.2 MV-DBMS

Multi Version Database Management Systems (MV-DBMS) employ MVCC schemes and manage versions of data items. MV-DBMSs differ from single version DBMSs (SV-DBMSs) that maintain a single version of a data item. SV-DBMSs lock data items on updates and perform updates in-place (overwrites). Therefore, the data item exists only in one version and there is no overhead of finding the visible version of a data item. Related approaches that also manage a multi-versioned dataset, such as Temporal DBs, are listed in Chapter 8.

MV-DBMSs are conceptually a good match for the properties of modern storage technologies such as Flash. Reads are never blocked and concurrent transactions are capable to leverage parallelism.

The design and algorithms in traditional approaches, as implemented in current databases, address special properties of HDDs - especially their high sequential throughput and high latency access time on any type of I/O. They maintain clustering by performing in-place updates to optimize for follow-up read accesses, reducing the latency introduced by the *mechanical* properties of HDDs (rotational delay, positioning time).

Concurrent updates are handled by the concurrency manager. This work focuses on Snapshot Isolation, which either employs the *first-committer-wins* or *first-updater-wins* rule (explained in more detail in Section 4.2.1). An important property of the MV-DBMS is the *invalidation model*. It defines how invalidation of updated/deleted versions is handled and has direct influence on the created I/O pattern. This is described in more detail in Section 4.1.

3.3 Multi Version Concurrency Control on SSD

MV-DBMSs, that implement MVCC schemes, are gaining more importance recently. They introduce significant performance improvements compared to approaches that implement protocols which require read lock. Especially the level of concurrency is increased, since read operations do not block write operations and vice versa. Concurrent access can be leveraged to exploit the Flash storage's inherent parallelism. An immediate effect is that locking overhead is reduced, but reads are more complex: due to version management it has to be ensured that the correct tuple version of a single data item is read. This matches the SSD's property of fast random and sequential reads.

In this work Snapshot Isolation (SI) is used as an exemplification of the MVCC protocols. The invalidation scheme in SI creates in-place updates caused by the invalidation of the updated tuple, which possibly increase the (random) write load. In addition, it generates mixed load since read and write locations intersect. In-place updates, small random writes as well as mixed loads negatively impact not only the performance of the Flash device, they also impact their endurance (Chapter 2).

The SI issues we discussed so far are: (i) *mixed load* in terms of read/write and (ii) *in-place updates*, caused by the in-place invalidation scheme.

Those issues are addressed by the SIAS approach. SIAS alleviates the necessity of the on-tuple invalidation since the presence of a successor version implicitly invalidates its predecessor. Thus it orders the versions in a simple backwards connected logical chain. The load is not mixed since a write is always performed as an append. Furthermore, once data is written it is immutable.

The concepts and approaches presented in this dissertation are implemented in PostgreSQL. They can be applied and transferred to other arbitrarily chosen MV-DBMS implementations. PostgreSQL is chosen due to its availability and acceptance in both industry and academia. It is open source and allows the implementation of the concepts and approaches presented in this dissertation.

4 Version Management

Contents

4.1 Invalidation Model **44**
4.1.1 One place invalidation 44
4.1.2 Two place invalidation 45
4.2 Snapshot Isolation **45**
4.2.1 Snapshot Isolation - Introduction 45
4.2.2 Snapshot Isolation - Algorithm 49
4.3 SIAS: Snapshot Isolation Append Storage **51**
4.4 SIAS-Vectors **52**
4.4.1 SIAS-Vectors - Data Structures 52
4.4.2 SIAS-Vectors - Access Paths 54
4.4.3 SIAS-Vectors - Discussion and Evaluation 56
4.5 SIAS-Chains **65**
4.5.1 SIAS-Chains - Algorithm 65
4.5.2 SIAS-Chains - On-Tuple Information 66
4.5.3 SIAS-Chains - Data Structures 67
4.5.4 SIAS-Chains - Access Methods Insert - Update - Delete 71
4.5.5 SIAS-Chains - Discussion and Evaluation 73
4.6 SIAS Garbage Collection **86**
4.7 SIAS Buffer Management **87**
4.8 SIAS Recovery **88**
4.9 SIAS Discussion - Worst Case Scenario **88**

Version management is essential to the MV-DBMS. Inserts create initial versions of data items and updates insert new tuple versions of an existing version. On a request for single or multiple data items the correct (visible) version of each item has to be found and the MV-DBMS has to execute a visibility check of the tuple versions that belong to the requested data items.

Management of multiple concurrent transactions is provided by the multi-version concurrency control mechanism (MVCC). There exist different approaches to MVCC [105] such as *Timestamp Ordering*, *Multi-version Two-Phase Locking* and *Snapshot Isolation*. Snapshot Isolation (SI) is a particular type of concurrency control scheme [105] and has gained wide acceptance in commercial and open source systems (Chapter 8). Old, updated versions get invalidated to indicate the existence of a newer version. The invalidation model is a performance critical component and is further discussed in the following Section.

4.1 Invalidation Model

Traditional, single version DBMSs maintain a single version of a data item. On an update the item usually gets locked for reading/writing. Subsequently the item is updated in-place (overwritten). Therefore, the data item exists only in one version and there is no overhead of finding the visible version of a data item.

MV-DBMSs add one dimension to the DB's data items, maintaining them in different tuple versions (Section 3.2). Whenever a data item is modified a new tuple version is created. On an access to the data item the correct tuple version has to be determined. This is facilitated by invalidating the former tuple version using an invalidation model.

For instance, on an insert the initial tuple version of a data item is created, while on an update a successor version is created. Deletions can be handled as updates (special *tombstone* tuple version) or the recent tuple version is invalidated without a successor.

The newest uncommitted tuple version of the data item may only be *visible* to the transaction that created it. Other transactions must access the most recent committed tuple version of the data item, i.e. the tuple version that was committed before the start of that very transaction. MV-DBMS implement mechanisms to identify the visible tuple version of a data item for a running transaction. At most one tuple version of the data item is allowed to be visible to a transaction.

There exist two basic invalidation schemes:

- *i) one place invalidation* (*valid from*),
- *ii) two place invalidation* (*validity range*).

In the following approaches to the invalidation model are explained utilizing the SI concurrency control scheme. Figure 4.3 depicts the two place invalidation scheme of the Snapshot Isolation approach. The one place invalidation scheme of the Snapshot Isolation Append Storage (SIAS) approach is depicted in Figure 4.7 (SIAS-Vectors) and in Figure 4.22 (SIAS-Chains).

4.1.1 One place invalidation

One place invalidation marks each tuple version with one timestamp. In general there is a choice of two variants on that timestamp: either a *validity* timestamp that corresponds to the logical moment in time from which on it is valid or a *creation* timestamp that corresponds to the logical moment it was created - usually the begin timestamp of the creating transaction. One place invalidation is a *write optimized* approach, since other (older) tuple versions of the data item do not have to be changed. For example, on an update of a tuple version simply a new version is created. This approach is in line with the append storage principle and does not require an on-tuple invalidation. However, this approach is optimal for operations that often create new tuple versions but seldomly read.

On each read access the visible tuple version has to be found. The choice of the timestamp determines the type of *visibility* checks that have to be executed in order to find the visible tuple version. This requires to fetch the most recent committed tuple version which was not inserted by a concurrently running transaction. Since tuple versions only have one *valid from* or creation timestamp, all tuple versions of a specific tuple have to be fetched and their timestamps have to be compared.

4.1.2 Two place invalidation

Two place invalidation assigns each tuple a *logical range* that defines its validity. On an update of a tuple version, a new tuple version is created and the predecessor version has to be invalidated. This requires an in-place update of the predecessor's visibility meta-data. Tuple versions are marked with a range, facilitated by two timestamps that form the visibility meta-data. Analogously to the one place invalidation scheme, different timestamps can be utilized. The first timestamp defines the logical moment in time from which on it is valid, or when it was created. The second timestamp marks the tuple version as invalid. It is set upon the invalidation of the tuple version by the creation of the successor version (or a deletion). Either when the successor version became valid or was created.

Two place invalidation is a read optimized approach, since the visibility decision can be made without fetching other tuple versions. All information that is needed to decide upon visibility of a tuple version is contained on that very version. When reading an arbitrary tuple version the transaction is capable to determine the visibility by checking the validity range stored on the tuple version. Therefore not all tuple versions of a tuple (data item) need to be fetched. On an update this approach inevitably has to update the validity timestamp of the predecessor, thus resulting in an in-place update.

In the following the term *data item* is used to describe a tuple and the term *tuple version* is used to describe a specific version of that data item. On symmetric storage additional effort is made to maintain a clustering of tuple versions in order to optimize for reading access (minimize I/O access times).

On asymmetric storage such clustering turns out to be counterproductive, writing in random is very slow and immediate overwriting is impossible (erase before write cycle in erase block granularity).

4.2 Snapshot Isolation

4.2.1 Snapshot Isolation - Introduction

Snapshot Isolation (SI) is an established multi-version concurrency control (MVCC) mechanism - introduced in [4]. It is commonly accepted and widely spread in both academia and industry. Some MV-DBMS are (no particular order): Oracle, PostgreSQL, Microsoft SQL Server, MySQL/InnoDB, Maria DB, Netezza.

SI is based on transactional timestamps, it assigns a (logical) timestamp to each transaction, whereas a data item's tuple version is assigned two different timestamps (two place invalidation). The transaction's timestamp corresponds to the start of each transaction and is unique for each transaction. The tuple version's timestamps correspond to the creation and invalidation of that version; their values correspond to the creating/invalidating transaction's timestamp. Invalidations are issued on an update/deletion.

The principle of SI is that each running transaction executes against its own version/snapshot of the committed state of the database and is isolated from effects of other concurrently running transactions. This allows a transaction to read an older committed version of a data item instead of reading a newer, uncommitted version of the same item or being blocked/aborted. A snapshot describes the visible set of item versions the transaction is able to "see", facilitated by the timestamps.

Operations of a transaction are isolated from concurrently running transactions. Whenever a data item is accessed the transaction’s timestamp is compared with the timestamps on the tuple version(s) of that item. A transaction is only allowed to read the most recent tuple version of a data item that was committed before the transaction started. Versions with a higher creation timestamp (inserted after the start of the transaction) are invisible and those with a lower (or equal) timestamp are visible to the transaction as long as they are committed and were not inserted by a concurrently running transaction. Tuple versions that have been inserted concurrently by still running transactions are only visible to the transaction that inserted them.

Reads are never blocked by writes and changes made by a transaction are executed on its own snapshot which becomes visible to follow up transactions after its successful commit. Whether a commit is successful or not is determined by the transaction manager. This is explained in the following paragraph.

Handling of Updates

The transaction manager handles concurrent updates either by using the *first-committer-wins* or the *first-updater-wins* [4, 104] rule.

The first-committer-wins rule is enforced during the commit time of a transaction. As the name suggests, the transaction which first finishes has the right to commit. This is an optimistic scheduling, since checks are deferred to the commit time. At the commit time the transaction manager performs a write set comparison of the involved concurrent transactions. Overlapping write sets between concurrent transactions are not allowed and lead to the abort of at least one transaction since it is not allowed to have more than one update to an item.

The first-updater-wins rule has to check each time a write/update is issued. It is implemented in PostgreSQL using exclusive locks. At most one transaction at a time is allowed to update a data item and only the youngest committed tuple version of a data item may be updated. If one transaction attempts to update a data item which is updated by a concurrent unfinished transaction, it waits until that transaction finishes. If the transaction finishes successfully (it commits), the waiting transaction has to abort, otherwise the waiting transaction can attempt to take the lock on the data item.

Example

Figure 4.1 depicts the history of three transactions. It illustrates an example where transaction T_1 creates the initial version X_0 of data item X with an attribute value of 9. X is subsequently updated by T_2 which creates tuple version X_1 with attribute value 10. Finally T_3 creates X_2 with attribute value 11. T_1 commits before T_2 starts and T_2 commits before the start of T_3. In the SI approach, T_3 modifies the timestamp attributes of X_2 and X_1.

$$\textbf{History:}\quad \underbrace{W_1[\mathbf{X_0=9}];C_1;}_{\text{Transaction } \mathbf{T_1}}\quad \underbrace{W_2[\mathbf{X_1=10}];C_2;}_{\text{Transaction } \mathbf{T_2}}\quad \underbrace{W_3[\mathbf{X_2=11}];C_3;}_{\text{Transaction } \mathbf{T_3}}$$

Figure 4.1: History: transactions T_1,T_2,T_3 execute after each other. T_1 inserts data item X in its initial tuple version X_0. T_2 updates X, creating tuple version X_1. T_3 subsequently updates X, creating tuple version X_2.

The resulting relation is depicted in Figure 4.2. The left side of the picture depicts the *outside* view on the relation. Only data item X is visible, the MV-DBMS abstracts from the versioning underneath. The right side depicts the MV-DBMS's view on the relation. It manages three tuple versions of data item X.

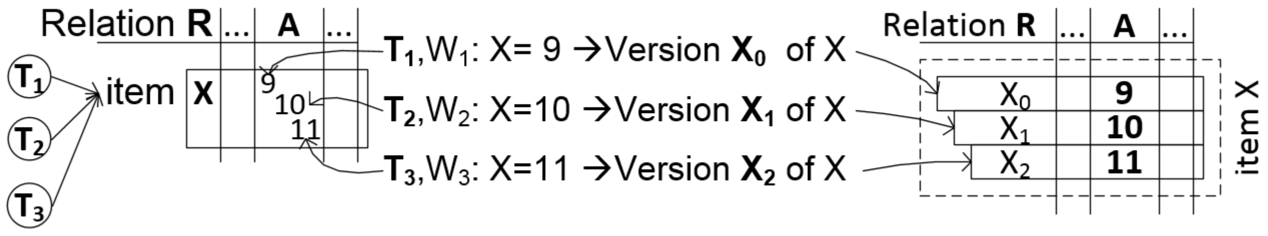

Relation R
...
A
...
item
X
9
10
11
T1
T2
T3
T1,W1: X= 9 →Version X0 of X
T2,W2: X=10 →Version X1 of X
T3,W3: X=11 →Version X2 of X
Relation R
X0
9
X1
10
X2
11
item X

Figure 4.2: Relation R contains data item X in three tuple versions: X_0, X_1, X_2. The MV-DBMS abstracts from the multi-version scheme. Each running transaction is only allowed to see at most one tuple version of X.

The resulting version organization and invalidation scheme is depicted in Figure 4.3. X_0 is updated in-place by setting the invalidation timestamp and X_1 is created. The creation timestamp of X_1 equals the invalidation timestamp of X_0 and pointers are set to form a doubly linked list. Analogously the update of T_3 creates X_2 and invalidates X_1.

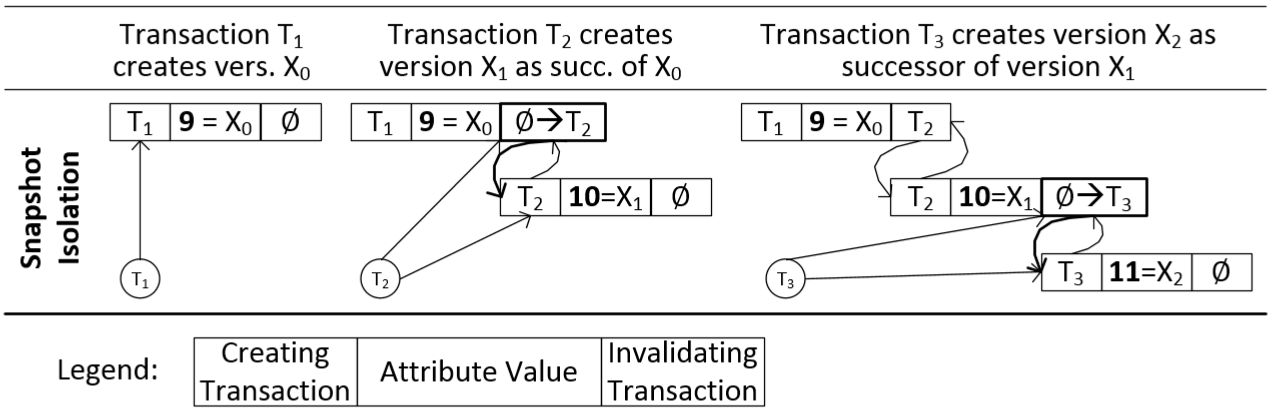


Figure 4.3: SI: invalidation scheme and logical version organization. T_2 updates X, creates tuple version X_1 and invalidates X_0 by setting the invalidation timestamp of X_0 to its own starting timestamp. This creates a small in-place update. T_3 proceeds analogously creating the second update (third tuple version X_2) of data item X.

Version Organization

SI assumes a *logical* version organization as a doubly linked list as well as a two point invalidation (depicted in Figure 4.3). Invalidations are issued on updates and deletions. The *physical* version organization is independent from the SI algorithm. In SI a new version is stamped with a *creation timestamp* and an old version is invalidated with an *invalidation timestamp* equal to the timestamp of the inserting/updating transaction. The double linked list, an optimization for HDDs in order to save read access time, forms a physical chain and is built by storing a pointer to the new version on the invalid version and vice versa. This is beneficial for symmetric storage since it enables determining the appropriate tuple version of a data item at the scope of the tuple version and involves less complexity on reads, e.g. in favor of sequential scans. When using single point invalidation all tuple versions of a data item have to be found in order to decide upon visibility.

In the physical version organizing scheme tuple versions get organized, (often) clustered and persisted on storage in order to speed up a follow up sequential read (scan) access. For instance, PostgreSQL favors to store the successor tuple version on the same page where the older version resides. In addition, PostgreSQL uses the *heap only tuple (HOT)* optimization in order to cluster tuple versions that belong to the same data item on the same page. These properties create physical small in-place updates which are particularly unsuitable for Flash. This can be seen in the evaluation in Section 4.4.3 (Figure 4.13).

Write Amplification

With MV-DBMS (SI) on Flash storage, there exist several places where write amplification can occur [68, 49, 114]. When a single attribute (net data) of a data item is updated in a MV-DBMS the remaining attributes stay the same.

- *On-tuple write amplification*: beneath the net data the update of a single attribute adds unchanged attributes.
- *Visibility information write amplification*: tuple versions store visibility information according to the version organization.
- *On-page write amplification*: in SI a new version is created out-of-place. Since the smallest I/O granularity equals a database page, header and footer information is added as page metadata.
- *DB I/O write amplification*: the whole page has to be written.
- *File system write amplification*: the structures of the file system are updated.
- *On-device write amplification*: on the Flash storage the usual overhead of the address mapping, garbage collection and page migrations have to be made.

In the following the write amplification (WA) that is caused by the invalidation scheme and the version organization is considered. On the invalidation of a tuple version the *visibility information* such as the timestamp is set. This involves an update to the page where the tuple version resides. In the worst case this creates an in-place update of the size of a whole page although only a much smaller timestamp is changed. The default db-page size in PostgreSQL is 8KB, whereas the invalidation timestamp is 32bit. Version organization related WA also occurs when tuple versions are clustered. Free-space per page/block is maintained to cluster tuple versions by attribute or version. If a successor of a tuple version is clustered on the same page as its predecessor, all other tuple versions that reside in that page have to be written as well.

Figure 4.4 shows the physical view for the example history as depicted in Figure 4.1. The initial tuple version X_0 of data item X is inserted by T_1 and gets placed on page P_0. On the first update to X, issued by T_2, the initial version is invalidated in-place. The whole page (8KB) receives a write request although only the timestamp value was set (32bit). The successor tuple version X_1 is placed on a different page (P_5). When X_1 is updated by T_3 the successor tuple version X_2 is placed on P_3 and analogously X_1 is invalidated in-place. This example leads to five page-write requests where two pages have to be updated in-place.

As an optimization for HDDs a successor tuple version could also be stored on the same page as the predecessor, yet this requires free-space reservation and free-space management for each page. This entails write amplification itself which depends on the amount of reserved space.

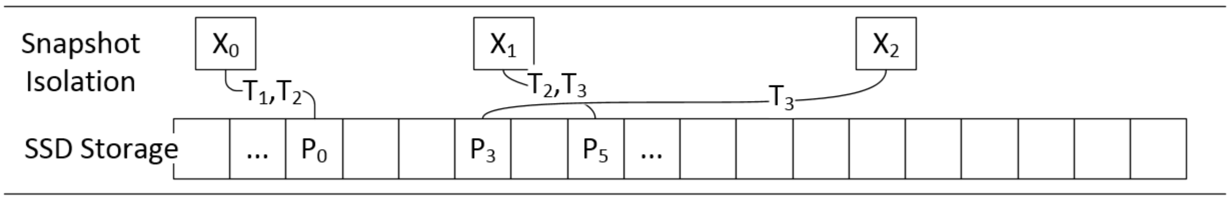


Figure 4.4: SI write amplification: Physical View on the SSD. The initial tuple version X_0 of data item X, inserted by T_1, is placed on page P_0. The first update to X, issued by T_2, invalidates X_0 in-place and the whole page receives a write request. X_1 is placed on P_5. T_3 updates X, invalidates X_1 in-place and creates X_2 which is placed on P_3.

4.2.2 Snapshot Isolation - Algorithm

In the following the SI algorithm, depicted in Algorithm 1, is described [59, 93, 105]. According to the principles of SI, each transaction tx receives a timestamp as it starts, and each tuple version X_n is stamped with a creation timestamp ($X_n.xmin$) and an invalidation timestamp ($X_n.xmax$) . The creation timestamp corresponds to the transaction that created/inserted the tuple into the database while the invalidation timestamp is set during the creation of a successor version by the transaction that inserts the successor. An update is handled as an insert of a new version X_{n+1} of data item X and the invalidation of data item X_n. To invalidate X_n, $X_n.xmax$ is set by the transaction that commits the update. The invalidation results in an update of the page in which the invalidated tuple version X_n resides. The update changes the version-information but not the contents of the tuple (data-load) and the page has to be written to stable storage in-place. The new tuple version may be stored on any page with enough free space. Optimizations may group new versions of a transaction on the same page [34], or on the page that contains the updated tuple. In the latter case there has to be a free space area reserved on each page, which results in poor space utilization and higher porosity.

Algorithm 1 illustrates the general SI, which for technical completeness is complemented with PostgreSQL specific details. In the following we use the notation of X_n for a data item X in version n; tx for a transaction with timestamp tsi; $X_n.xmin$ for the creation timestamp and $X_n.xmax$ for the invalidation timestamp of a given tuple version X_n. Note that $X_n.xmin$ and $X_n.xmax$ are equal to the timestamps of the inserting or invalidating transactions respectively. Complementary to the tuple versions, PostgreSQL maintains a Snapshot Data structure ($SnapData$) for every running transaction tx. $SnapData$ enables visibility checks and write-set comparison. Among other fields it contains: (i) $SnapData.xmax$ - the (cutoff-) transaction ID of the next transaction ($tx_{(+1)}$ at the time tx started) which serves as a visibility threshold for transactions whose changes are not visible to tx; (ii) $SnapData.xmin$ - determines transactions whose updates are visible to tx and denotes the transaction ID of the lowest still running transaction; (iii) $SnapData.xip$ - holds a list of all transaction IDs concurrent to tx. Finally, PostgreSQL uses a main memory structure called PG_CLOG (in short Log), based on the database log, which allows for fast transaction status checks (aborted, committed, in progress).

Algorithm 1 Snapshot Isolation Algorithm

```
 1: StartTX($tx$) tsi=timestamp ($tx$);
 2: procedure READ($tx$, $X_v$)                                      ▷ $tx$ reads tuple version $X_v$
 3:     if $X_v \in \{writeSet(tx)\}$ then
 4:         READOWNVERSION($X_v$,$tx$)
 5:     else
 6:         READSTABLEVERSION($X_v$,$tx$)
 7:     end if
 8: end procedure
 9: procedure WRITE($tx$,$X$)                                         ▷ $tx$ modifies X
10:     if (VERSIONCHECK($tx$,$X$) $\leftarrow Failed$) then
11:         ROLLBACK($tx$)
12:     else
13:         $tx.lockX \leftarrow$ REQUESTANDWAITFORLOCK($tx$,$X$)
14:         if ($tx.lock \leftarrow granted$) then                     ▷ Perform Update
15:             $X_s \leftarrow$ READSTABLEVERSION($tx$,$X$);
16:             $X_i \leftarrow$ NEWVERSION($X$); $X_s.xmax = tsi$;
17:             $X_i.xmin = tsi$; $X_i.xmax$=NULL;
18:         else                                         ▷ another transaction has a lock on X
19:             ENQUEUELOCK($tx.lockX$)                                ▷ $tx$ waits
20:             if ($tx.lock \leftarrow granted$) then WRITE($tx$,$X$);
21:             end if                                                 ▷ Re-Check
22:             ON Ti.commit() or Ti.rollback(): UpdateLog;
23:             Release aquired Locks; WakeUp waiting Transactions;
24:         end if
25:     end if
26: end procedure
27: procedure READSTABLEVERSION($tx$,$X$)
28:     Find $X_a$ created by $tx_k$ such that:
29:     ($X_a.xmin < tx.SnapData.xmax$) AND ($LOG(X_a.xmin) == committed$)
30:     $AND((X_a.xmax == NULL)$ OR $(LOG(X_a.xmax) == aborted)$ OR $(X_a.xmax \in \{tx.SnapData\}))$
31:     if $checks \leftarrow FAIL$ return $NULL$ end if
32:     return $X_a$
33: end procedure
34: procedure VERSIONCHECK(X)
35:     $X_i$ = READSTABLEVERSION($X$)
36:     if ($X_i$==$NULL$) return $FAILED$ end if
37: end procedure
```

Example: Consider the read of an item X by transaction tx. Tx reads either its own version (line 4) or another, committed and stable version in the DB (line 27). A *stable version* has to be visible to tx which means it has to be committed before tx started (line 29). At this point $X_v.xmax$ indicates the existence of a successor to tuple version X_v. By checking $X_v.xmax$, tx is able to determine if a successor version exists and if so, to decide if that version should be read instead. Considering an update of a data item (line 9). Transaction tx has to perform a version check (line 34) to check if the tuple version that is about to be updated is the stable version. Under *first-updater-wins*, tx requires a write lock on the tuple version to successfully perform the update. The tuple version could be locked by another concurrent transaction, in which case tx waits until it is unlocked (line 19). On unlock of the tuple version, tx has to check whether the tuple version is still the stable version (concurrent transaction aborted or created no successor version) or if a new successor tuple version was created (tx aborts).

In summary: the invalidation timestamp related update is necessary for the SI algorithm since the tuple versions are accessed as individual entities and not as multiple tuple versions of a single data item. The in-place invalidation has to be avoided on SSDs since it reduces endurance and hampers performance.

4.3 SIAS: Snapshot Isolation Append Storage

We analyzed SI (Section 4.2.1) to develop a better suitable Flash-aware concept. Along with the investigation of different log/append based storage managers (further details on LbSMs are given in Section 6) we developed the Snapshot Isolation Append Storage (SIAS) concept.

The SIAS approach subsumes changes on the algorithmic and architectural design of the MV-DBMS with the focus on asymmetric Flash storage. Several components that were designed on the performance properties of HDDs are reconsidered and evaluated for their effectiveness on Flash storage.

SIAS introduces a new conceptual invalidation model and changes the paradigm of addressing tuple versions individually. SIAS addresses tuple versions that belong to the same data item as a single structure. The conceptual changes allow the exploitation of append based storage management that operates on tuple version granularity (rather than pages). The append operation is conceptually realized in tuple granularity and is physically implemented for a block device interface. This design decision makes the approach applicable to current device hardware layouts. Since the block device interface is not a conceptual assumption, the approach is also applicable to smaller (or larger) granularities, e.g. it can be applied to byte addressable NVMs.

The buffer management is adapted and simplified: to be applicable to the block device interface a write retention is implemented. New tuple versions are logically appended to a buffered page that is written when it is completely filled or a threshold is reached (time, fill-grade of a page). Once a page has been appended it is immutable. Access methods such as scans and indexes are augmented by the SIAS approach.

First approximations of the potential of LbSM are based on simulation (Chapter 6). The basic concepts are integrated into the SIAS-Vectors prototype which further analyzes the potential of SIAS and led to the development of the SIAS-Chains concept. The evaluation of the results of the SIAS-Vectors implementation confirms and validates the analyzed potential of the initial simulation. SIAS-Chains is evaluated on different Flash storage, such as USB Flash storage, single SSDs and Flash SSD software RAID. Details and evaluation of the simulation are contained in Chapter 6.

First *SIAS-Vectors* is presented in Section 4.4. SIAS-Vectors implements the basic SIAS concept and shows the benefits of tuple granularity appends, the revised invalidation scheme and version organization. It demonstrates the advantages of the *lower levels* of the approach. Yet the transaction management and access paths are not yet optimized in the realization. SIAS-Vectors only delivers limited support for index access paths and incurs overhead on scans.

The improved approach *SIAS-Chains* is presented in Section 4.5. SIAS-Chains comprises further conceptual changes of the invalidation model and the version organization along with the access methods (Section 5) and visibility checks.

4.4 SIAS-Vectors

This Section introduces the *Snapshot Isolation Append Storage Vectors* (SIAS-Vectors) approach. SIAS-Vectors is the initial prototypical implementation of the SIAS algorithm in PostgreSQL as demonstrated in [33]. It conceptually introduces a new version organization and invalidation model: The SIAS-Vectors invalidation scheme conceptually omits the invalidation timestamp for each tuple version. SIAS-Vectors eliminates visibility meta-data related in-place updates on asymmetric storage and employs tuple granularity LbSM. SIAS-Vectors applies a position based identification of tuple versions through the utilization of bitmap vectors that code validity and visibility.

4.4.1 SIAS-Vectors - Data Structures

First the SIAS-Vectors data structures are discussed. SIAS-Vectors uses bitmap vectors to indicate different states of a single tuple version, depicted in Figure 4.5.

- B_1 is the *validity bit-vector*. It indicates the invalidation of each tuple version. If the *bit is set*, the corresponding tuple version is invalid and a valid successor is contained in the database. A tuple version is considered invalid if and only if it has a valid (committed) successor. If the *bit is unset* the tuple version is valid and serves as the *entrypoint* (most recent valid version) of the data item. B_1 is mandatory and logically partitions the database into valid and invalid tuple versions.

- B_0 is the *dead tuple version bit-vector*. If the *bit is set* the tuple version is invisible (dead) to any running transaction due to the visibility rules of SI. B_0 is optional and represents an optimization.

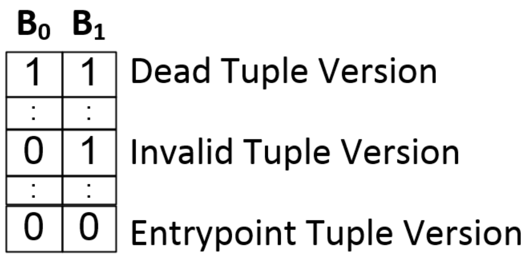


Figure 4.5: SIAS-Vectors Data Structures: Mandatory Invalidation Bitmap Vector B_1. Optional invisible (dead) tuple version Bitmap Vector B_0.

A bit-vector is a lightweight data structure, since only one bit per tuple version is required. On an invalidation of a tuple version X_n of data item X, the creation timestamp of the following version X_{n+1} denotes the invalidation timestamp of its predecessor X_n. The access paths are modified such that accesses to a data item use the most recent tuple version. Either the most recent tuple version has to be read, or an older tuple version has to be fetched by traversing the singly linked list. As soon as the visible version is found or the initial version has been reached, the traversal is interrupted. The detailed visibility check of SIAS-Vectors is described in Algorithm 2 and explained in Section 4.4.2.

On-tuple Information

SIAS and SI store information on each tuple version that is used to determine the visibilty. The tuple structure of SIAS and SI is depicted in Figure 4.6. SIAS-Vectors stores no version invalidation information of the current tuple version on the corresponding tuple. The invalidation information is coded in the version chain of the data item, such that a tuple version X_{n+1} stores its own creation timestamp $X_{n+1.create}$ while the invalidation timestamp equals the successor's creation timestamp $X_{n.create}$.

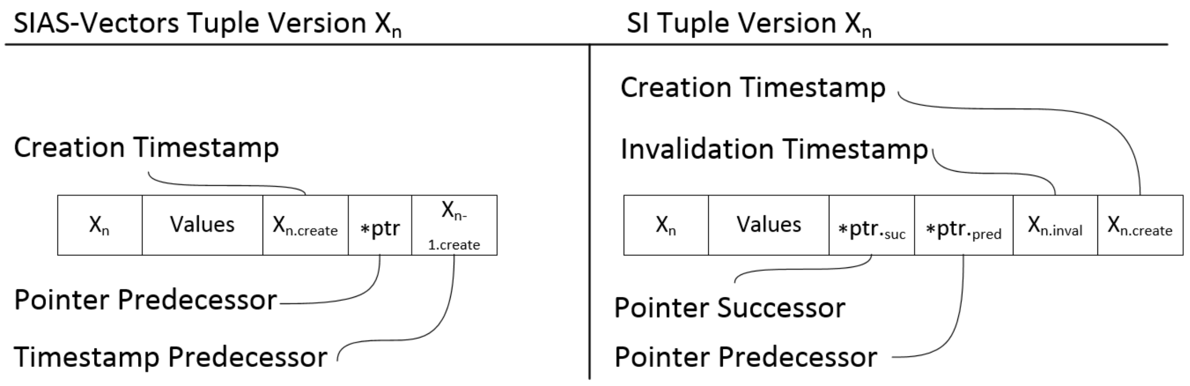


Figure 4.6: SIAS-Vectors and SI on-tuple visibility information. SIAS-Vectors stores no invalidation information on each tuple version itself. Each tuple version stores the creation timestamp of a predecessor tuple version.

Example

Reconsider the history from the example history in Figure 4.1. Figure 4.7 depicts the invalidation using SIAS-Vectors. Each tuple version's state is indicated within the bitmap vectors. After T_1 created tuple version X_0, the validity bit-vectors are updated accordingly. T_2 creates an update to version X_0: the invalidation bit is set within B_1. After the commit of T_2 no running transaction is able to access (see) X_0, hence the visibility bit is set in B_0. This also classifies the tuple version to be garbage collected. T_3 creates and appends X_2 and sets the bit values in the bit vectors B_0 and B_1. Assuming that X_1 is still visible to *some* running transaction(s), the bit in B_0 is unset.

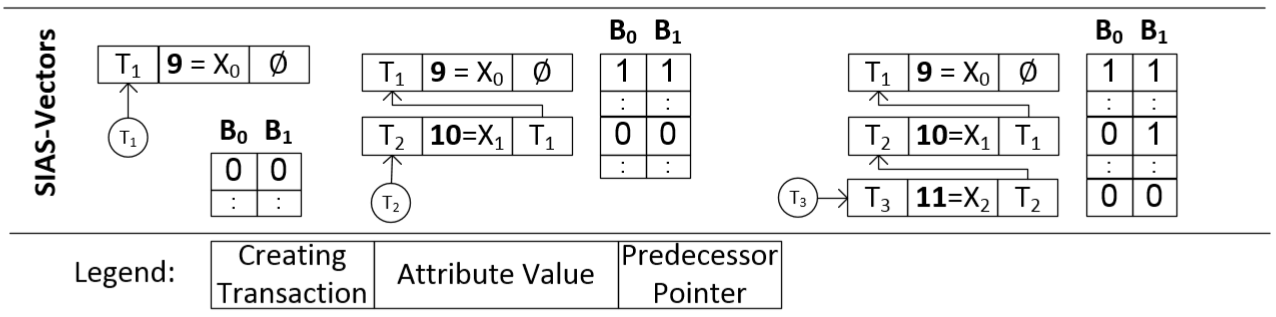


Figure 4.7: SIAS-Vectors: B_0 indicates dead (invisible) tuple versions and B_1 indicates invalid tuple versions. X_0 is inserted and becomes the *entrypoint* of the data item. T_2 updates X and inserts X_1. X_0 is invalidated by setting the bit in B_1. X_1 becomes the *entrypoint*. T_2 commits and the bit is set in B_0 since X_0 is a dead tuple version. T_3 updates X, invalidates X_1 by setting the bit in B_1 and inserts X_2 which becomes the new *entrypoint*.

4.4.2 SIAS-Vectors - Access Paths

Read

On the execution of a scan a virtual snapshot of B_1 is created, representing the state of the relation at the time the scan starts. The virtual snapshot is created by using the *memcopy* operation. B_0 indicates versions that can be discarded (garbage collected) and is not copied since changes to it depend on the oldest still running transaction. False negatives (not visible) are avoided, but false positives may exist, which are filtered during the visibility check using B_1. For instance a valid tuple version may be invisible for a running transaction if that transaction has to read a previous tuple version.

Each tuple version maintains a back-link to and the creation timestamp of its predecessor (see Figure 4.7). From each entrypoint the visibility check, depicted in Algorithm 2, is executed. If the check returns false there can exist a visible predecessor version X_{n-1} to some tuple version X_n. Tuple version X_{n-1} is fetched using the pointer on X_n and can subsequently be checked. No other visible successor version exists, due to the local copy of B_1, thus line 11 can never be reached on scans.

A fetch of the successor version is the most costly operation for SIAS-Vectors. This case can only occur when accessing an invalidated tuple version directly, for instance on an index access. It is the worst case, since the successor version's position in the database is not known. On a call of *getSuccessorVersion()* the relation has to be searched for the entrypoint of the data item. This operation is not necessary in the SIAS-Chains approach (Section 4.5), since the entrypoint of a data item is always known.

Update/Insert:

An update consists of an invalidation of the old and an insertion of a new version. To invalidate the old version, the respective bit in B_1 is set and the new version with its corresponding values in both vectors (initial zero values) is inserted. The new version receives its own creation timestamp, maintains a pointer to its predecessor and the predecessor's creation timestamp. The SIAS-Vectors buffer management applies write retention: The tuple version is appended to a buffered page which is written after it is completely filled or a threshold is reached.

Algorithm 2 SIAS-Vectors Visibility Check [33]

```
1: procedure VisibilityCheck(TupleVersion $X_v$, Transaction $tx$)
2:     if ($X_v.txmin == tx_{tid}$) then
3:         return $true$;                                   ▷ tx reads its own version
4:     else if (!$B0[X_v.id]$) then                         ▷ check if garbage version is dead
5:         if ($X_v.txmin > tx_{id}$) then
6:             return $false$;
7:         else if ($X_v.txmin < tx_{id}$) then
8:             if (!$B1[X_v.id]$ AND $isCommitted(X_v.txmin)$) then
9:                 return $true$;
10:            else
11:                Tuple $X_{new} = getSuccessorVersion(X_v)$;
12:                if ($X_{new}.txmin < tx_{tid}$ AND $isCommitted(X_{new})$)
13:                return $false$; else return $true$; end if
14:            end if
15:        end if
16:    end if
17:    return $false$;
18: end procedure
```

Figure 4.8 depicts the physical view of the tuple append in SIAS-Vectors. Tuple versions of a relation are buffered and appended to the head of the LbSM. Compared with Figure 4.4 this shows the potential of the write reduction. Instead of writing three pages, only one page is appended. Traditional approaches to logging, such as write ahead logging and recovery are not affected (Section 4.8).

Section 4.4.3 discusses the potential of the write reduction. Details on LbSMs are contained in Section 6.

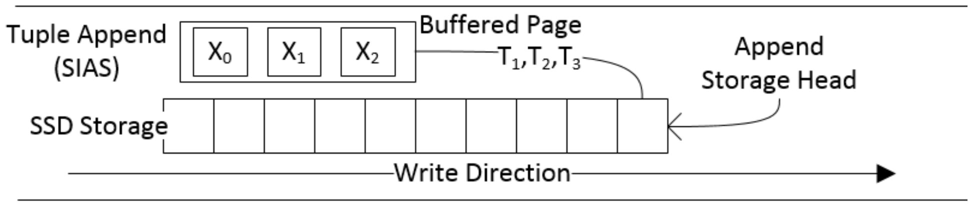


Figure 4.8: Write reduction of the SIAS concept, physical view. The buffer manager applies write retention. New tuple versions are appended to a page that is held in the buffer. When the page is completely filled or a threshold is reached, it is appended to the head of the LbSM.

Delete:

A deletion is treated as an invalidation and an insertion of a tombstone version terminating the version chain. The tombstone version becomes the *entrypoint* and when it becomes the only visible version, the bits are set in B_1 and B_0 to render all versions of the data item invisible (same indication as a dead tuple version) - no access to the tuple is needed in order to discard it during the visibility check.

4.4.3 SIAS-Vectors - Discussion and Evaluation

The tuple append functionality of SIAS-Vectors was initially simulated in order to focus on the I/O characteristics of the basic SIAS concept. Figure 4.9 depicts the evaluation process of the SIAS-Vectors approach. After the initial simulation of the algorithm the potential benefits were analyzed. Details of the simulation process are also described in Section 6.4. The simulation has been utilized to determine parameters such as the granularity of the append and the choice of data placement. The simulation was also used to estimate possible benefits of the SIAS concept. After the potential of the SIAS concept was established, the SIAS-Vectors algorithm was implemented in PostgreSQL. The prototypical implementation has been tested using the same benchmark configuration in order to cross validate the results of the simulation. The resulting write patterns are equivalent with and comparable to those collected during the simulation and the implementation shows the actual effects. In the following the write pattern is discussed with the focus on the write reduction. The effects of the write reduction are analyzed using the TPC-C benchmark and a micro benchmark.

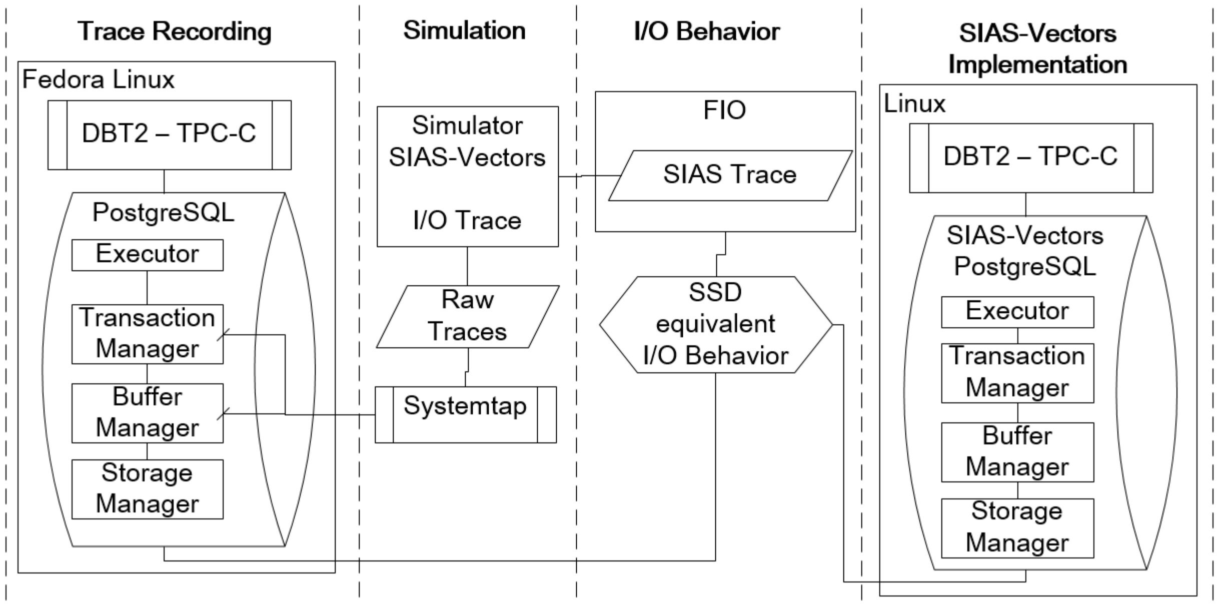


Figure 4.9: SIAS-Vectors evaluation. Initial results are based on a trace driven simulation. The simulator creates I/O patterns according to the SIAS concept. The SIAS-Vectors prototype is executed on the same SSD and the I/O patterns are compared.

Write pattern.

The write pattern of SIAS-Vectors and SI is recorded using blktrace [10] to demonstrate the effects of the tuple append operation and to emphasize the write reduction. The traces have been recorded during the demonstration of the SIAS-Vectors approach in [103, 33]. TPC-C has been instrumented with a low number of one warehouse and very short runtime of 360 seconds in order to better demonstrate and visualize the resulting access patterns.

The graphs in Figure 4.10 and 4.12 represent written block numbers. Each dot represents a page write of 8KB. Figure 4.10 depicts the SIAS-Vectors write pattern for the customer and stock relation of the TPC-C benchmark. It illustrates the append operation. The block numbers are offset oriented, each relation begins with block number zero. The customer graph's scale begins at block number 6800. Because of the append operation new tuple versions are inserted at new blocks which are buffered and appended when completely filled, hence there are no write requests below that block number. Analogously the stock graph's block number begins at 13000. In-place updates are avoided. Once a page is written it is considered immutable, except for later garbage collection (reclaim space). Figure 4.12 and Figure 4.11 show the same benchmark configuration with the traditional SI algorithm, developed for HDDs. In-Place updates are scattered across the whole relation, beginning with block zero. New inserted data shows as a line at the top of the graph.

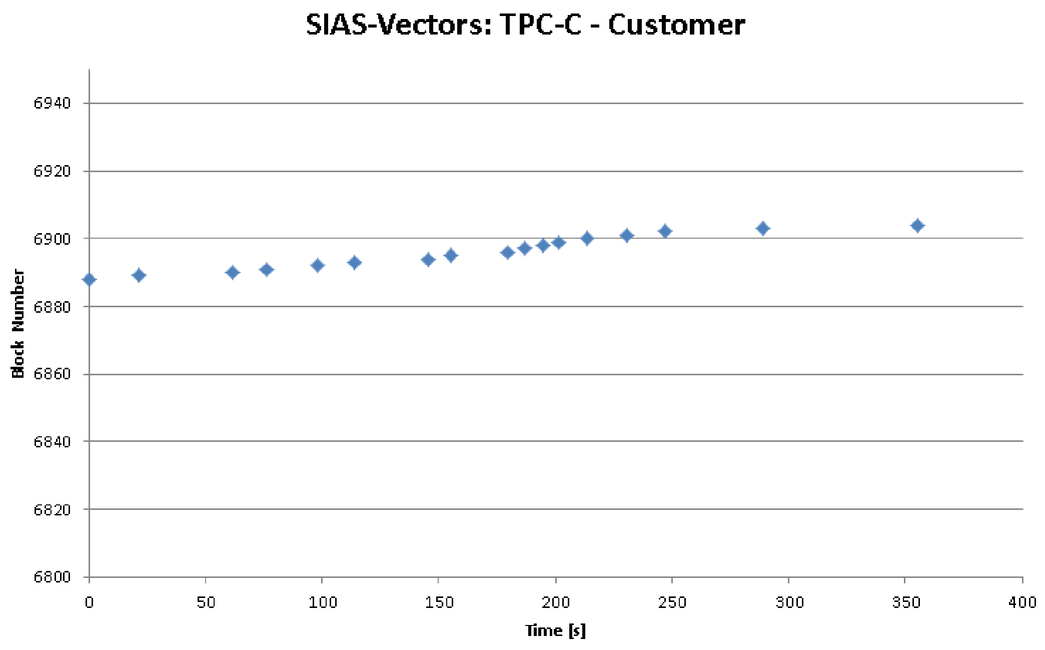


Figure 4.10: SIAS-Vectors. Write pattern of the customer relation. Each dot represents a page write of 8KB. Tuple versions are appended to a buffered page which is written when it is completely filled. Block numbers are offsets.

Write reduction

Compared with the SI in-place update concept Figure 4.4, the tuple append (Figure 4.8) yields potential to for the reduction of issued writes. Instead of writing three pages, only one page is appended.

Each dot in Figures 4.10, 4.11, 4.12 and 4.13 represents a page write of 8KB. Figure 4.10 and Figure 4.12 show the write access to the customer relation of SIAS-Vectors and SI. They illustrate the significant write reduction of the SIAS approach. The stock relation is depicted in Figure 4.11 and Figure 4.13. Using write retention new tuple versions get appended to a page which is held in the buffer until it is completely filled or a threshold is reached (Figure 4.8). In the customer relation SIAS-Vectors appends 17 pages which sum up to 136KB, respectively 53 pages (424KB) are appended to the stock relation. In comparison, the SI approach has to write and update 386 pages (approx. 3MB) in the customer relation and 2566 pages (approx. 20MB) in the stock relation. SIAS-Vectors writes the net amount of new inserted tuple versions. SI updates tuple versions in-place, thus updating pages that also contain other tuple versions which have

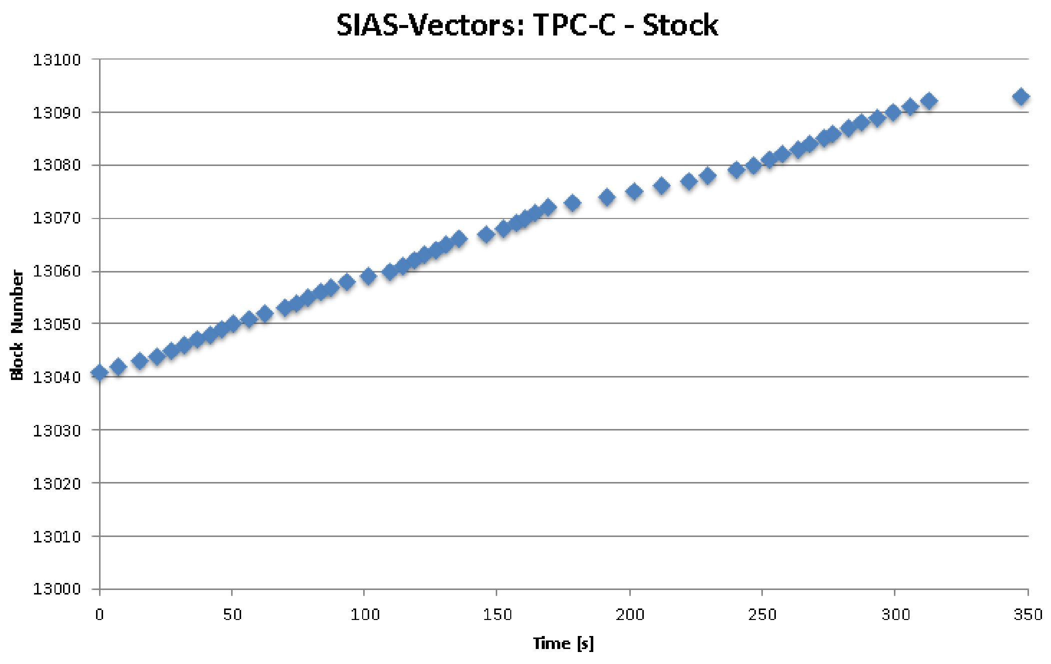


Figure 4.11: SIAS-Vectors. Write pattern of the stock relation. Each dot represents a page write of 8KB. Tuple versions are appended to a buffered page which is written when it is completely filled. Block numbers are offsets.

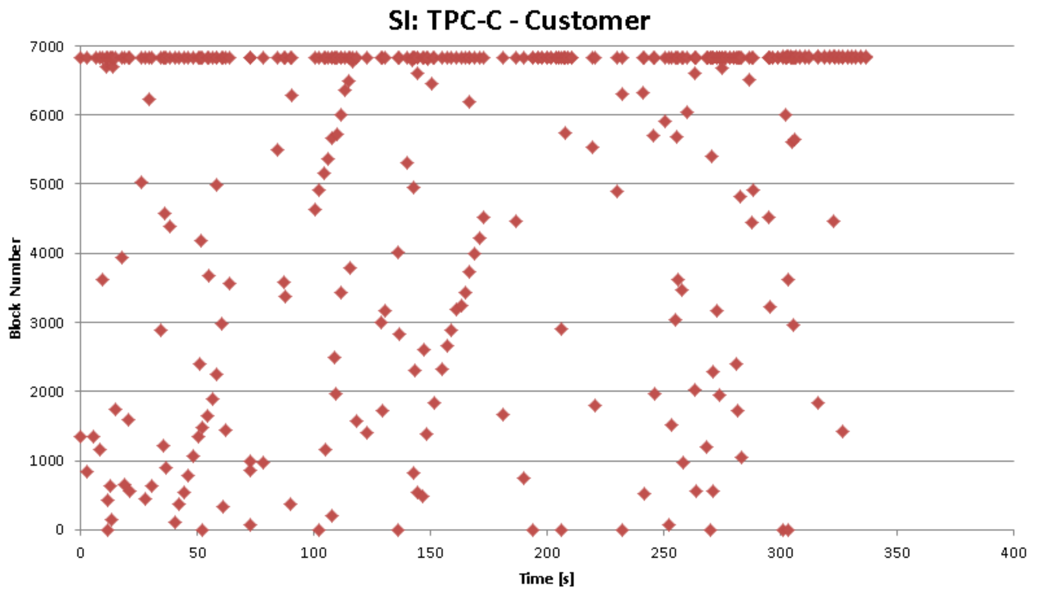


Figure 4.12: SI. Write pattern of the customer relation. Each dot represents a page write of 8KB. In-place updates and writes are scattered along each relation. Block numbers are offsets.

to be written again. Obviously this increases the load on the I/O storage system in terms of read and write operations and incurs additional buffer space consumption and eventually cache pollution.

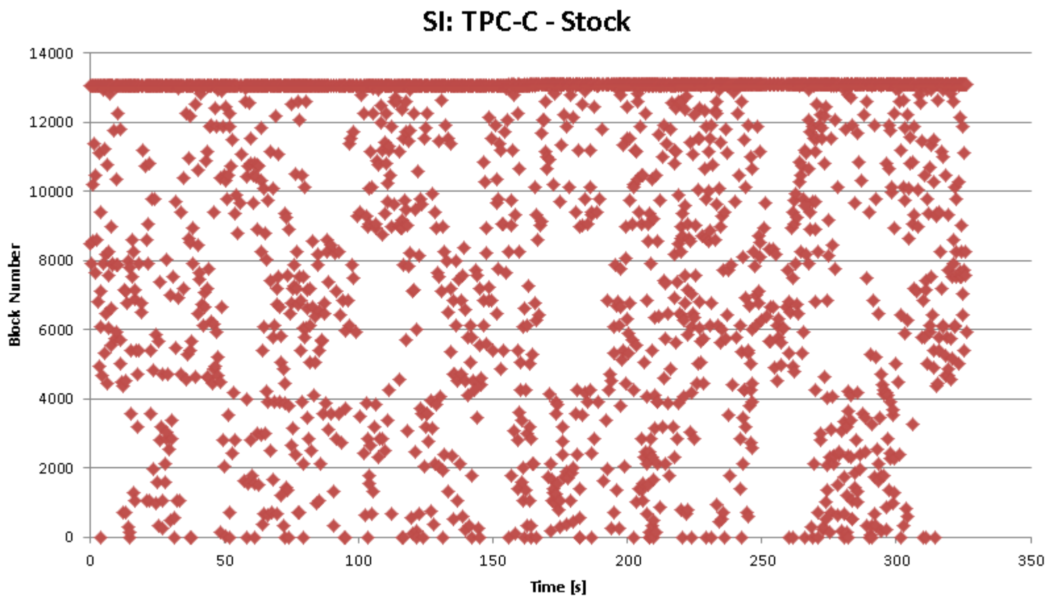


Figure 4.13: SI. Write pattern of the stock relation. Each dot represents a page write of 8KB. In-place updates and writes are scattered along each relation. Block numbers are offsets.

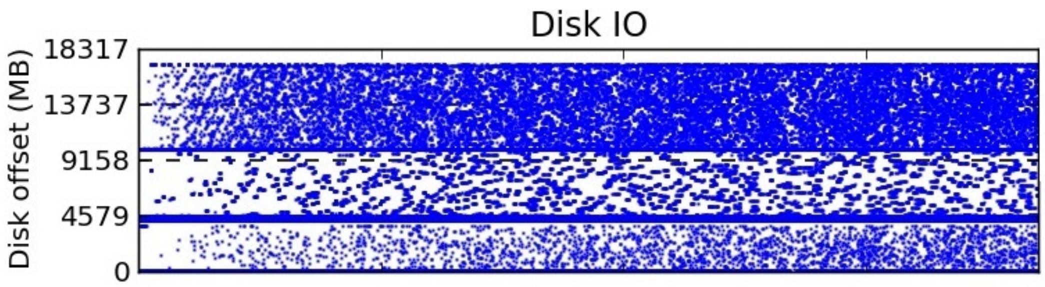


Figure 4.14: Simulated Write Pattern: SI with in-place storage management. All relations of the TPC-C benchmark are monitored - Figure also contained in Section 6.3.

Figure 4.14 shows the write pattern of the simulation of the SI approach and Figure 4.15 shows the write pattern of the SIAS-Vectors approach. Both Figures are also contained in Section 6.3. The same parameters of the simulation are realized in the SIAS-Vectors approach, which are the tuple LbSM and individual write LbSMs for each relation. The Figures depicts the *disc I/O* for write accesses. For SIAS-Vectors the *disc I/O* block trace shows the same write lines, just for *all* TPC-C relations, as in Figure 4.10. Analogously the *disc I/O* block trace of the SI in-place storage management shows the same scattered writes among each relation.

Micro Benchmark - One Tuple Update

The *one tuple update micro benchmark* is designed to show that SIAS-Vectors exhibits low response times per transaction on write-intensive and/or mixed workloads and delivers stable performance figures. It directly shows the effect of the write reduction. Response times under SI tend to exhibit higher workload dependent variance. In this benchmark exactly one data item in each page of the relation is updated. The amount of concurrent transactions is increased. The standard deviation is higher on SI than it is on SIAS. It implies the following: if a relation

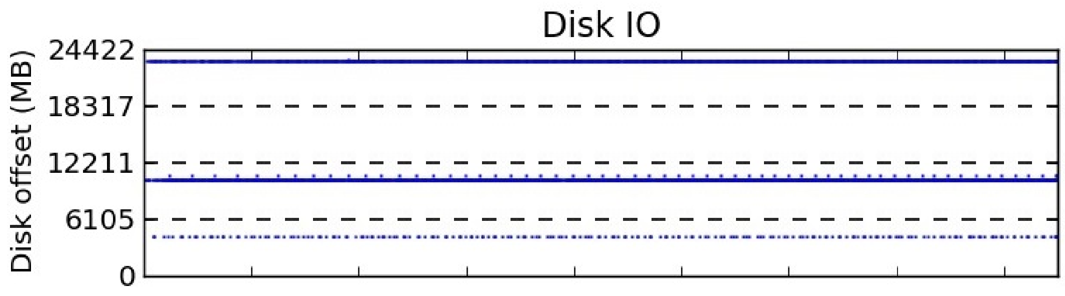


Figure 4.15: Simulated Write Pattern:SIAS tuple LbSM. All relations of the TPC-C benchmark are monitored - Figure also contained in Section 6.3.

comprises 100 pages and a page contains approx. 200 tuple versions, SI will update 100 pages and writes them back in-place in order to mark each tuple version that is updated as invalid (visibility information). This results in multiple random writes and in addition it creates a single new page for the new tuple versions. Conversely SIAS-Vectors needs one page, which is buffered until it is filled, to which the new tuple versions are appended. This benchmark corresponds to the worst case update predicate touching only a sparse subset of the relation. For instance in real world workloads this corresponds to a budget increase of all departments that do not exceed a certain maximum budget. The result of this micro benchmark are depicted in Figure 4.16 with deactivated write cache and in Figure 4.17 with activated write cache. It shows the runtime (msec.) for different benchmark runs as well as the standard deviations as error bars. It is executed on two different Flash SSDs, an Intel X25E and a Samsung 830 MLC SSD.

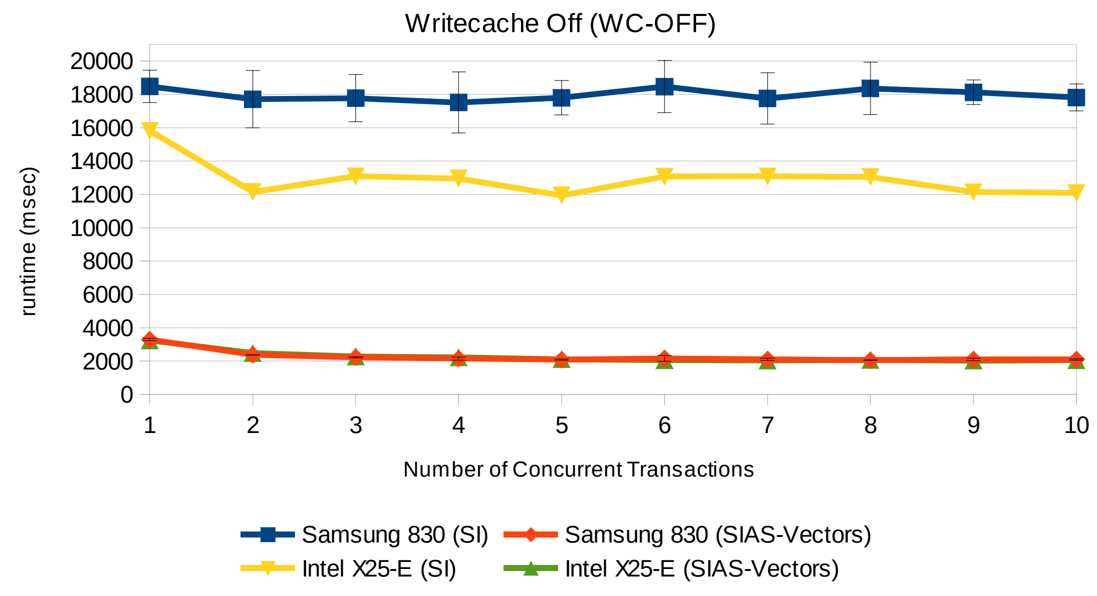


Figure 4.16: Micro Benchmark. One Tuple Update on SSD. Update exactly one data item in each page of the relation. Measures runtime, varies the amount of concurrent transactions. Write Cache Off.

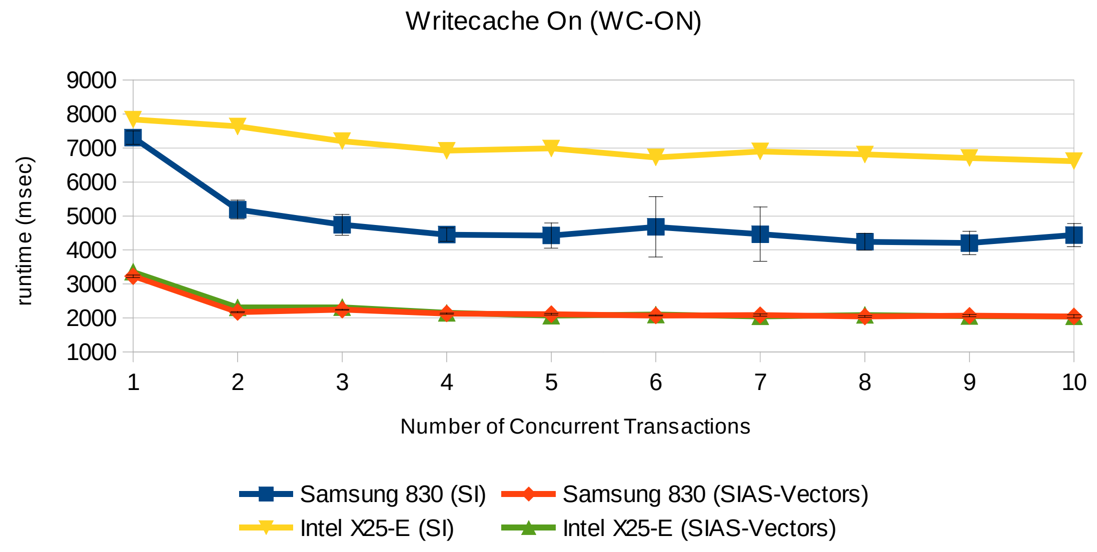


Figure 4.17: Micro Benchmark. One Tuple Update on SSD. Update exactly one data item in each page of the relation. Measures runtime, varies the amount of concurrent transactions. Write Cache On.

SIAS-Vectors also delivers better results on HDDs. This claim is demonstrated by executing the one tuple update micro benchmark on an HDD. The results are depicted in Figure 4.18 with deactivated write cache in Figure 4.19 with activated write cache. SIAS-Vectors delivers stable performance with low response/ completion times. The observed performance improvement is caused by the significant write reduction and the avoidance of random writes. Yet the improvement on SSDs is higher.

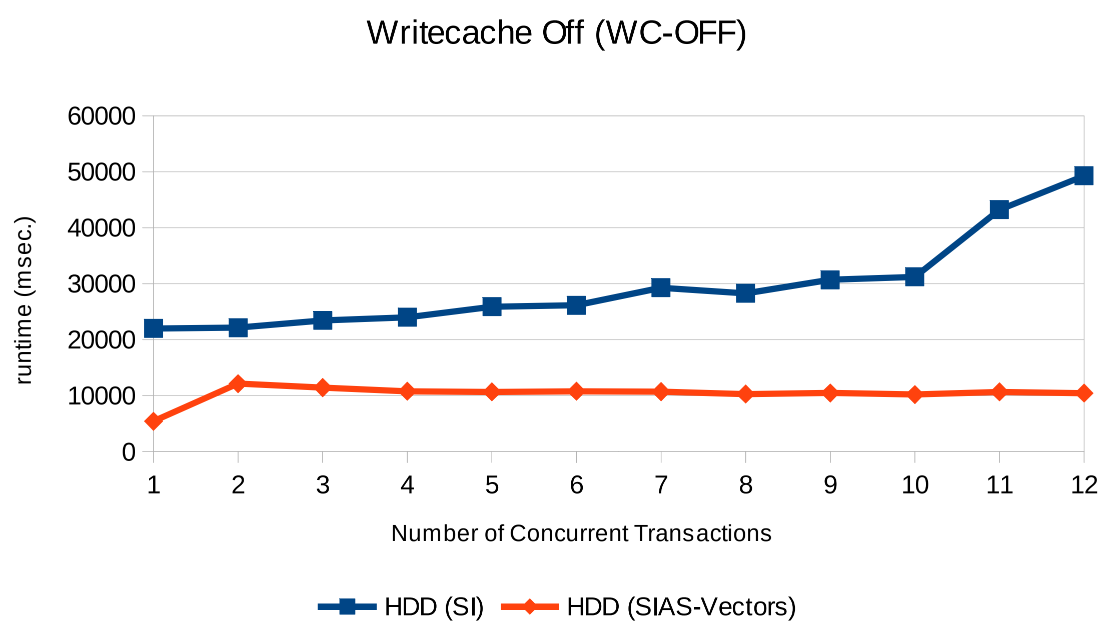


Figure 4.18: Micro Benchmark. One Tuple Update on HDD. Update exactly one data item in each page of the relation. Measures runtime, varies the amount of concurrent transactions. Write Cache Off.

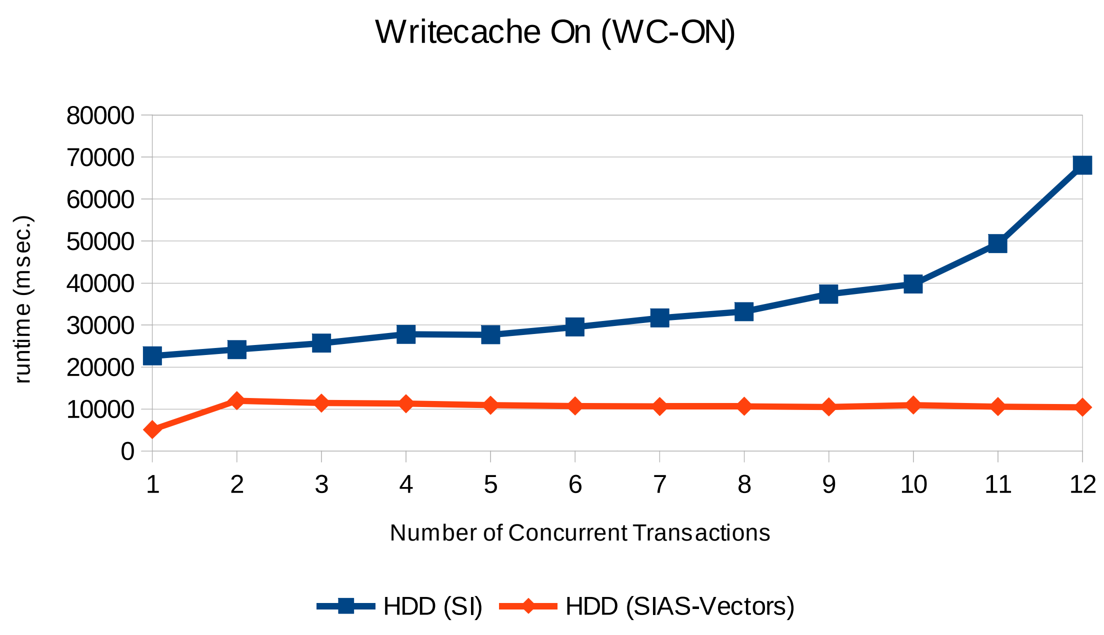


Figure 4.19: Micro Benchmark. One Tuple Update on HDD. Update exactly one data item in each page of the relation. Measures runtime, varies the amount of concurrent transactions. Write Cache On.

Length of the Version Chain

In our test-runs we observed the length of the chains and the amount of subsequent reads needed to receive the visible version. We assume that there are at most two more accesses necessary to find a visible version under TPC-C workloads. This assumption is confirmed by the test results and the actual number of average reads needed was smaller than expected. We chose a long running TPC-C test with 240 minutes runtime and a small dataset to increase the amount

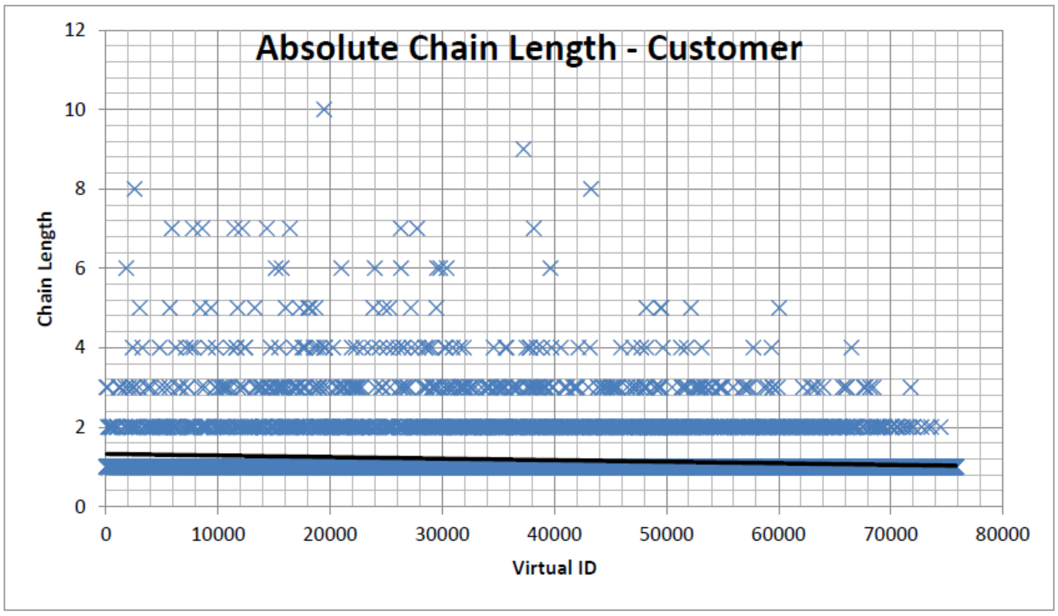


Figure 4.20: SIAS-Vectors. Length of the version chain after the execution of the TPC-C benchmark on the customer relation. Without garbage collection of dead tuple versions and space reclamation. Average chain length *customer*: 1.15. Average chain length *stock*: 2.18. The average length of *visible* versions in a chain is below 1.01 tuple version(s).

of updates to individual tuple versions. The following numbers consider solely the updated tuples of the respective relations (non-updated tuples have a chain-length of zero), those are taken out of the equation. Figure 4.20 and Figure 4.21 show the absolute tuple version chain length of the customer and the stock relation, respectively. In each chain that contains at least one update. The maximum amount of tuple versions for a data item is 10 for the *customer* and 75 for the *stock* relation.

The more important numbers are the *average tuple version chain length* as well as the length of *visible tuple versions* in the chain, since only the visible tuple versions can be accessed by a transaction. The visibility check (Algorithm 2) terminates before fetching older (invisible) tuple versions. Therefore, the maximum chain length is not relevant for checking. The average chain length of the *customer* relation was 1.15 and 2.18 tuple versions for the *stock relation*. The average length of visible tuple versions, which is the most important number, within a chain is below 1.01 but not lower than 1. This low number can be explained by the TPC-C workload. Most transactions (read and update) in TPC-C are fast and short running. In addition only one transaction is allowed to update a data item at a time. The absolute number of all visible tuples (including tuples with chain length of zero) is 600 032 for the *customer* and 1 999 992 for the *stock* table (PostgreSQL statistics). Note: In the case of other *mixed* workloads yielding long running reading transactions, the chain length can grow, yet other transactions that started recently do not traverse the whole chain. They stop at the most recent tuple version in the chain that is visible for them.

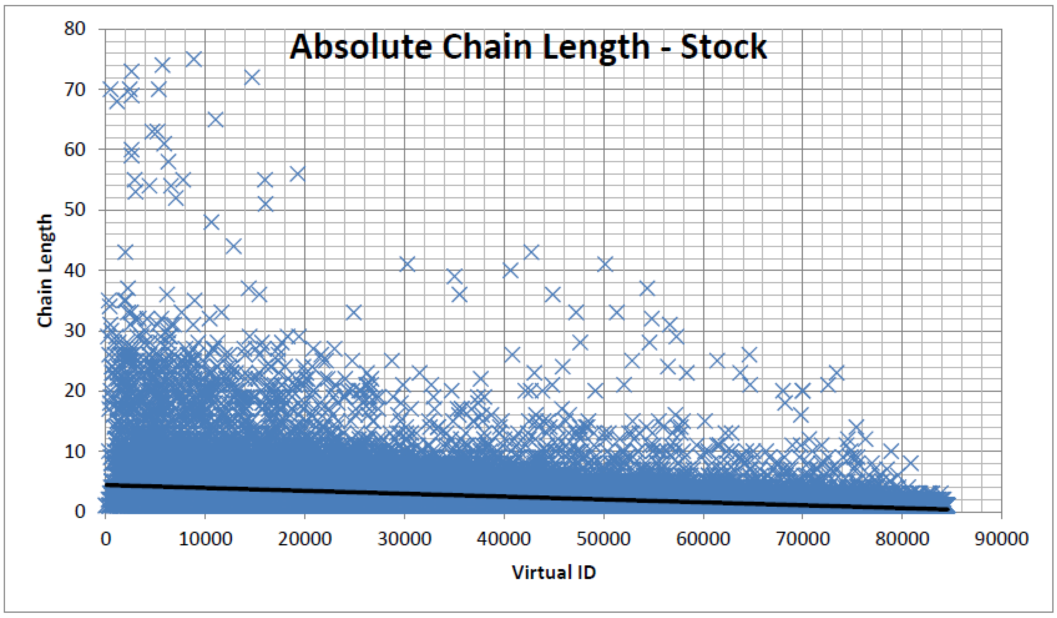


Figure 4.21: SIAS-Vectors. Length of the version chain after the execution of the TPC-C benchmark on the stock relation. Without garbage collection of dead tuple versions and space reclamation. Average chain length *customer*: 1.15. Average chain length *stock*: 2.18. The average length of *visible* versions in a chain is below 1.01 tuple version(s).

Benefits:

The benefits of the SIAS-Vectors approach are as following.

- SSD-friendly write patterns: SIAS-Vectors reduces random writes by utilizing out-of-place updates.
- Write reduction and endurance enhancements: SIAS-Vectors packs versions densely on pages and writes them sequentially, without the need of rewriting the page in which an old version resides. Tuple versions are appended to a buffered page which is written to a block on the SSD when the buffer is completely filled, or a (configurable) threshold is reached.
- Stable write throughput: SIAS-Vectors generates separate write/read streams and avoids mixed load. Each relation uses its own LbSM and appends to a separate region on the SSD.
- Low Response times for write intensive and/or mixed workloads.
- Lower scan times under read-only workloads due to the denser packing of tuple versions.

Known issues of SIAS-Vectors:

An existing issue of the SIAS-Vectors approach refers to the direct access to a tuple version that is *not* the entrypoint of the data item, e.g. through an index (primary or secondary). On such an access there might exist a visible successor version which has to be found. This path is coded in Algorithm 2, line 11. In the worst case this results in a scan of the whole relation. Therefore, an index would have to store information about the visibility as well.

Another issue is the overhead on scans: SIAS-Vectors creates a *memcopy* of the bitmap vector B_1, although this is a relatively fast mechanism.

Both issues are handled in SIAS-Vectors but incur overhead to the MV-DBMS. To correct these shortcoming the SIAS-Chains algorithm was developed. It is described in Section 4.5.

4.5 SIAS-Chains

This section introduces the *Snapshot Isolation Append Storage - Chains* (SIAS-Chains) approach. SIAS-Chains improves on SIAS-Vectors. As SIAS-Vectors it no longer follows the paradigm of addressing each tuple version individually. It improves on SIAS-Vectors by addressing tuple versions that belong to a unique data item as a set that belongs together: Tuple versions of a data item receive a unique identifier - *virtual ID* (VID), which is the same across all data item's tuple versions. As in SIAS-Vectors the successor of a tuple version stores a reference to the predecessor version's physical location. Hence a backward chain (singly-linked list) is constructed and in-place updates are conceptually avoided. The most recent (newest) tuple version of each data item is always known and called the *entrypoint* of the data item. The use of bitmap vectors is not required and the entrypoint does not have to be a committed tuple version (e.g. an update in progress). A more efficient data structure (Section 4.5.3) keeps track of the entrypoint of each data item. The mandatory data structure VID_{map} stores a mapping of the VID to the TID of the most recent version of the data item (entrypoint). In Section 4.5.3 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. Detailed examples are available in Sections 4.5.1 and 4.5.3.

SIAS-Chains key features are:

- a new invalidation schema;
- a new paradigm to version management and organization: addressing of data items using a VID that identifies all tuple versions belonging to the same data item;
- forming a *backwards directed chain of tuple versions*;
- appending of inserted/updated data using an LbSM;
- support of index access paths;
- appending in *tuple granularity* and writing densely filled pages;
- no page is allowed to be altered after it has been physically written (except for garbage collection);
- adaptation of the index access path using the data item abstraction VID.

4.5.1 SIAS-Chains - Algorithm

Figure 4.22 depicts an example of the SIAS-Chains invalidation scheme (same example history as in Section 4.2.1, Figure 4.1). In this example data item X of relation R is initially inserted in tuple version X_0 by transaction T_1. SIAS-Chains assigns data item X a unique identifier VID_x. Each access to a tuple version is served by the VID_{map} using the data item's VID. T_2 updates X_0 and creates the new tuple version X_1. T_2 sets VID_x to the TID of the new version X_1. X_1

receives a pointer to the physical location of the predecessor version X_0. Finally X is updated by T_3 which creates X_2, setting VID_x accordingly. The relation therefore comprises three different tuple versions of X. In comparison (referring to the example in Section 4.2.1, Figure 4.3) 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.

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 4.5.3) always points to the *entrypoint* of X, which contains information about the previous tuple version (if one exists). Each tuple version X_n is augmented with visibility information (Figure 4.23):

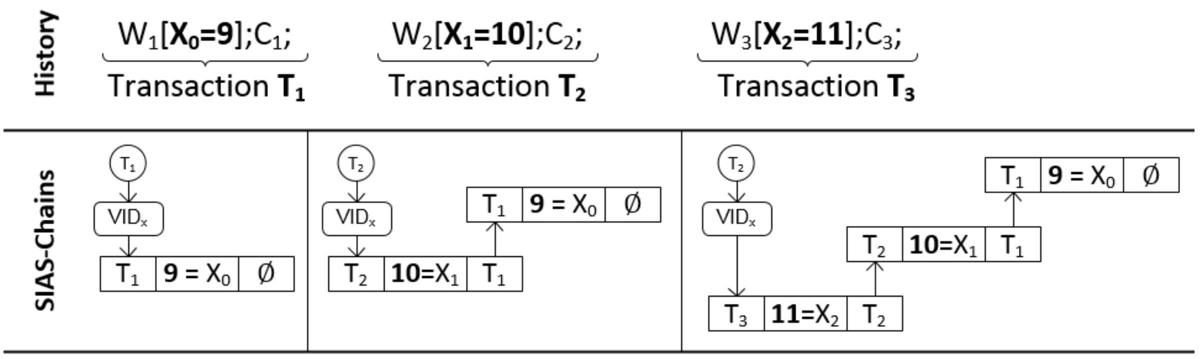


Figure 4.22: Example: SIAS-Chains invalidation (SI). Access methods in SIAS-Chains are served by the VID_{map} and follow the entrypoint of the data item.

4.5.2 SIAS-Chains - On-Tuple Information

Analogously to SI and SIAS-Vectors, each tuple version X_n in SIAS-Chains is augmented with the visibility information (Figure 4.23) in addition to general tuple attributes such as the attribute values.

- The creation timestamp $X_{n.create}$ which is denoted by the inserting transaction's ID.
- A unique VID which is equal among all tuple versions of the data item it represents.
- A pointer $*ptr$ which stores the physical reference to an existing predecessor version, or $NULL$ if no such version exists.
- The timestamp of the predecessor $X_{n-1.create}$.

There is explicitly no invalidation information stored on each tuple version. SIAS-Chains augments the SIAS-Vectors on-tuple visibility information (Figure 4.6) with the unique VID of the corresponding data item. The chained structure of the data item's tuple versions *code* this information along the version chain. The tuple versions of a single data item form a chronologically sorted chain. The creation timestamp of the following version equals the invalidation timestamp of its successor version, hence the visibility check methods of the original SI can be applied. In Figure 4.22 the *entrypoint* is X_2, after T_3 has committed. X_2 is *chained* to the predecessor tuple version X_1 which itself is chained to the initial version X_0.

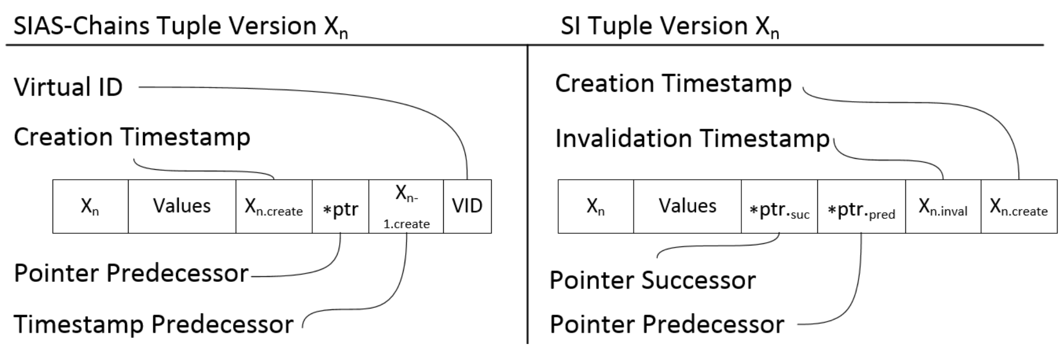


Figure 4.23: SIAS-Chains and SI on-tuple visibility information. SIAS-Chains augments the SIAS-Vectors on-tuple visibility information with the VID of the data item. SIAS-Chains stores no invalidation information on each tuple version itself. Each tuple version stores the creation timestamp of a predecessor tuple version.

Example: SIAS-Chains On-Tuple Information

Another example of the SIAS-Chains on-tuple information is given in Table 4.1, which is depicted in the logical relational view in Figure 4.24. The relation contains data items X and Y. X_0 was inserted by a transaction with the transactional timestamp 15. Since it is the initial version its predecessor pointer $*ptr$ points to itself and the creation timestamp of the predecessor $X_{0-1.create}$ is *null*. X receives the unique VID *0x0*. X is updated by transaction 38, which creates tuple version X_1 and sets $*ptr$ to X_0 and $X_{0.create}$ to 15. The VID_{map} points to the physical position of X_1, which is *0x123*.

Analogously Y was inserted by transaction 50 and received the unique VID *0x23*. Transaction 83 created the second version Y_1, set $*ptr$ and $Y_{0.create}$ accordingly and updated the VID_{map}. Transaction 110 creates the most recent tuple version Y_2, sets all fields analogously and updates the pointer in the VID_{map}. No predecessor tuple version is updated in-place. Which enables the LbSM to append immutable tuple versions.

Table 4.1: SIAS-Chains: On-Tuple Information

Tuple Version X_n	$X_{n.create}$	$*ptr$	$X_{n-1.create}$	VID
X_0	15	X_0	null	0x0
X_1	38	X_0	15	0x0
Y_0	50	Y_0	null	0x23
Y_1	83	Y_0	50	0x23
Y_2	110	Y_1	83	0x23

4.5.3 SIAS-Chains - 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, such as indexes and scans.

The VIDs are monotonously increasing positive numbers, which are unique for all tuple versions of the same 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

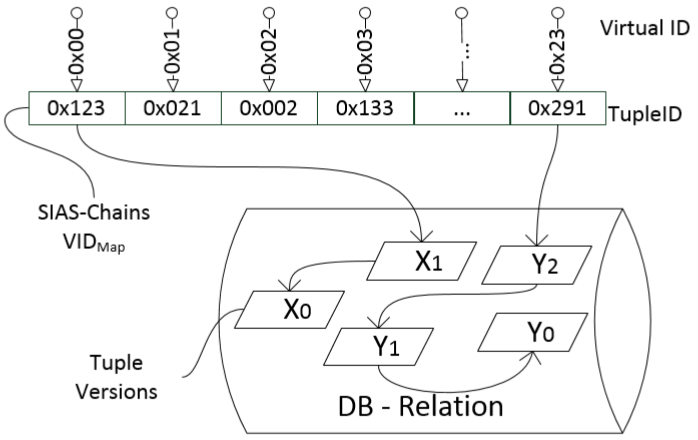


Figure 4.24: SIAS-Chains relational view of Table 4.1 with the VID_{map}. Tuple versions are connected with a backwards simply linked chain. The VID_{map} stores the TID of each data item's entrypoint.

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 VID 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 - its physical location. Pre-loading and bulk-loading can be supported, e.g. new VIDs 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) TIDs 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) within that block.
- iii) The maximum amount of TIDs that fit into a page is 1365, exclusive header.
- iv) We store a maximum of 1024 TIDs per page, a 10bit offset per TID is used.
- v) The record format is one TID (of the latest version, the entrypoint) per VID.

In the following the possible data structures for the implementation are discussed.

Array

The array is intended to be held in memory enabling direct access to TID values. The read and write costs are low, since each VID also serves as the index to the value in the array. Hence the reading cost C_R of a TID value is O(1). The writing costs C_W are the same O(1), yet for concurrent access the array position has to be latched. A downside to the in-memory array is the management and persistence of the structure. A static array requires the pre-reservation of space while a dynamically growing/shrinking array needs management overhead, e.g. pointers and handlers. Because of the in-memory architecture there is no *refined* page layout that could evict certain parts of the mapping. The static array method works best for small databases that do not have the need to evict the mapping. On a crash the mapping needed for the SIAS-Chains algorithm can be reconstructed using a sequential scan on each relation (can be piggy-backed during recovery).

Hashtable

On large DB sizes the mapping may not fit completely into main memory and therefore parts of it need to be swapped to persistent storage. In order to give the DBMS more control over the swapping process the VID_{map} is augmented with a page abstraction. The TID-filling degree of a page is 100%, as long as it is not the last page of the mapping. The bucket size equals the database page size. Since VIDs of a relation are sequentially assigned, the buckets also 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 collisions are impossible due to the ascending order of the VIDs. Each VID 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. Using this implementation queries on VID ranges are also facilitated.

An important part is the concurrency on the structure. On modifications of data items (operations that change or insert a TID that belongs to a VID) as well as on updates the corresponding storage slot within a hash bucket is latched. No latches are needed for read access, since the TID values are unchanged.

On the Hashtable there are two design decisions possible. Either a whole bucket or a single VID can be latched on updates (inserts). A separate bit-vector, that is correlated to the VIDs, can store latching information on VID granularity and serves as a VID latching table. Since each bucket contains 1024 TIDs, the bit-vector seems to be capable of better supporting concurrent access and therefore promotes parallelism. Instead of latching a whole bucket and subsequently blocking access to 1024 TIDs, a single bit is set for the VID.

Note: There is no *lock contention* on the VID_{map}. The *first-updater-wins* rule is applied and on modifications the tuple version is locked with an exclusive lock.

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: $2 * C_R + Latch$. The buffer has to be accessed and dirtied, hence twice C_R. The CPU denotes the cost to calculate the TID's position.

The SIAS-Chains implementation uses a fixed amount of slots that can be held in the cache for each relation. One slot accommodates one page containing one hash bucket. The amount is parameterized and can be chosen at configuration time. The pages are managed by the default PostgreSQL clock and sweep buffer management algorithm. All pages get flushed to disk when the database is shut down. Since all version information is contained on the tuple versions, there is no additional recovery logic for the VID_{map}. After a crash it can be reconstructed during the scan of the whole relation, which is done in PostgreSQL anyhow (piggybacked) - more details on recovery are contained in Section 4.8.

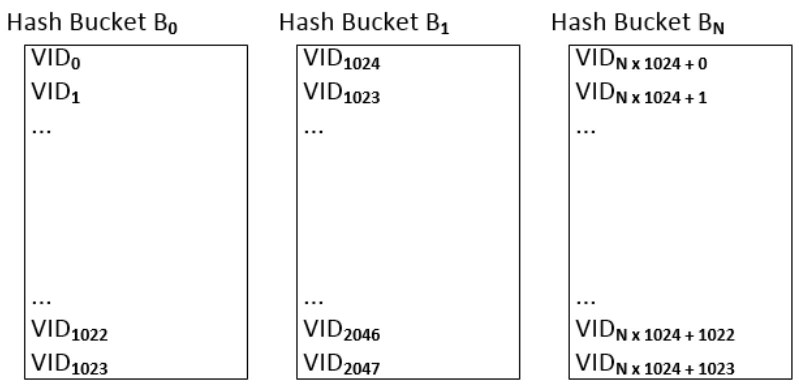


Figure 4.25: Hash Buckets for the VID_{map} with a capacity of 1024 TIDs. VIDs are monotonously increasing integers. VIDs are not physically stored, they represent the position of a corresponding TID.

Tree Structure

A tree structure, such as a B^+ tree, also seems to be a valid candidate for the VID_{map}. Since there are mostly point queries and range queries are also possible on the hashtable, due to the entropy of the VIDs (unique and monotonically increasing), the capability of range searches is not exclusive to the tree structure. For the SIAS-Chains approach an adapted tree structure is suggested that makes use of the VIDs entropy. Each leaf contains 1024 TID values and the position of a single TID does not have to be searched within a leaf node, since it can be calculated by a modulus of 1024 and the VID (similar to the hastable) if the tree is optimized such that the leaf nodes have the same format as the buckets in the hashtable approach. This serves as an optimization but requires a special handling of rotations in index nodes.

The access cost is comprised of the traversal through the index nodes to the leaf node, calculation of the position and the access of the TID: C_R = Traversal + CPU + O(1).

The update cost C_W in the tree structure is comprised of the traversal to the leaf node containing the TID, the latching/lockin, calculation of the TID's position within the leaf and the access plus the update of the leaf node: C_W=O(1)+O(1)+Traversal+Latch.

In addition operations have to be executed that involve rotations of index nodes (search key values), as soon as an index node "pops".

In terms of concurrency, the tree structure has to implement tree locking protocols ([78, 21]), since rotations may require to update index nodes. Locks and Latches have to be acquired in root to leaf order and have to be released in inverse order. Therefore the latching has to be made on a higher granularity which hinders concurrency, compared to the hashtable which is capable of latching a single VID.

The above mentioned special implementation of a tree structure, dedicated to the VID to TID mapping, involves special handling of index nodes, hence the following tree structure is suggested. It is especially designed for the requirements and has several performance optimizations and it is intended to be a minimalistic implementation. Instead of implementing default B+Tree conform rotations, the index nodes are constructed using the amount of TID entries (1024) that fit into a leaf page. Leaf nodes always contain 1024 subsequent VID-TID entries that are aligned on the $nx2^{10}$ border, where n denotes the corresponding leaf page (or bucket). Using that approach enables to calculate the exact position of each TID within a leaf page and not having to search. New VIDs are inserted into a leaf node with the capacity of 1024 en-

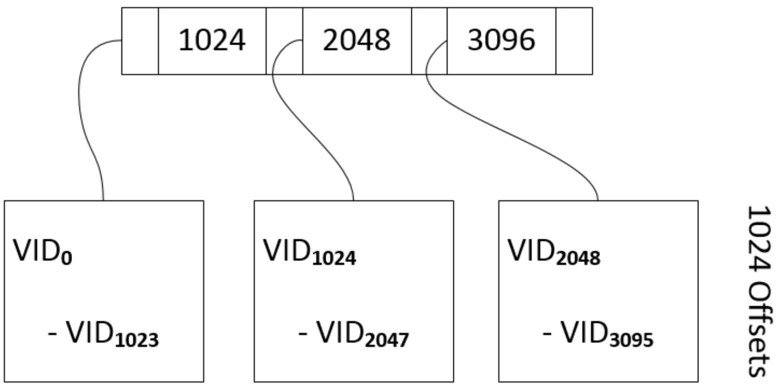


Figure 4.26: Tree Structure. VIDs are monotonously increasing and inserted in ascending order. Each leaf yields the capacity to store 1024 VIDs. Offsets can be calculated to directly access a leaf node.

tries, aligned at VID borders of $\{nx2^{10} - 1; nx2^{10}\}$; new index nodes are appended on the layers above. Values within leaf nodes are not rotated and the indexed search keys stay the same. This obviously creates an intermediate and temporally unbalanced tree structure but saves on rotation overhead. Intermediate index nodes can be reused and entries within a leaf are not rotated.

4.5.4 SIAS-Chains - Access Methods Insert - Update - Delete

The visibility rules for SIAS-Chains are equal to those of SIAS-Vectors. On an *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.

Algorithm 3 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:      $X_{0.create} = tx_{id}; X_{0.pred.create} = null;$
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
```

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 set analogously set to 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 physical 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 example in Figure 4.22 depicts this process. 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 1 line 7).

Algorithm 4 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. This is only necessary as long as there are running transactions, capable of *viewing* older tuple versions that have been recent before the deletion. 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. After those transactions have finished, the data item is not visible to any running transaction anymore. In that case a simple $NULL$ value can be inserted into the VID_{map}, hence removing the pointer to the entrypoint of the data item. The remaining tuple versions can be garbage collected, archived, compressed, etc..

4.5.5 SIAS-Chains - Discussion and Evaluation

The SIAS-Chains algorithm is implemented in the PostgreSQL [93] database system. We compare it to the original PostgreSQL code base, utilizing the DBT2 [22] open source implementation of the TPC-C [113] benchmark. In the following benchmarks and tests the PostgreSQL system relations, e.g., catalog and meta-data are stored separately on SSD for all experiments to exclude PostgreSQL specific management overhead.

The algorithms are evaluated on different Flash storage media, in order to emphasize the effects of the I/O patterns generated by the prototype. An artificial I/O bottleneck is created on the system by storing the tablespace of the database on a *Sandisk 16GB USB Flash drive*.

The *USB stick* instantly exemplifies the specific properties of the Flash chips. It also allows to discuss the effects of SIAS-Chains on a relatively small dataset and delivers a clear visualization of the I/O patterns. This is comparable to a system using Flash storage that becomes the I/O bottleneck to the remaining hardware components.

Although SSDs employ several optimizations and management processes, the properties of the Flash memories can only be *"masked"* up to a certain point which is reached as soon as the cache gets full or the over provisioning area is used up (Chapter 2).

This effect is shown in the TPC-C benchmarks on *SSD Flash storage*. The TPC-C performance benchmarks are conducted on an enterprise class SLC NAND Flash SSD (Intel X25-E 64GB SLC SSD). This delivers comparability to the preceding TPC-C benchmarks for SIAS-Vectors and the initial simulation of the LbSM.

In addition we executed the TPC-C performance benchmark on a small SSD RAID system. Two SSDs are connected by a software RAID that comprises two Intel X25-E SSDs using mdadm (Linux software RAID [70]). The RAID is built in performance level 0 (stripe) over the full capacity of the devices.

The following Sections argue that the I/O patterns of the different Flash storage media illustrate transferability and deliver comparable results. They are in-line with the results of the previously conducted benchmark runs for SIAS-Vectors and the previously conducted simulations.

Effects of the SIAS-Chains algorithm are recorded and visualized using the following tools: *blktrace* [10] is instrumented to record the I/O patterns; *blkparse* [9] is used to extract the amount of issued I/O requests out of the traces recorded by blkparse; *iowatcher* [51] generates the graphs to visualize the recorded traces.

I/O Pattern

In SIAS-Chains write accesses are realized as append operations for each relation (local append region - Section 6.3). The blocktrace in Figure 4.27 depicts the I/O pattern on the USB Flash storage, recorded during a TPC-C benchmark with 14 warehouses and a 2 hour runtime. Each relation appends new pages containing fresh tuple versions to a local append region (green dots).

The scan operation over the VID_{map} leads to more selective access which creates random read operations, visible as blue dots between the offsets 512 and 873. Since the dataset is relatively small, once fetched pages stay cached. The comparable I/O pattern confirms the

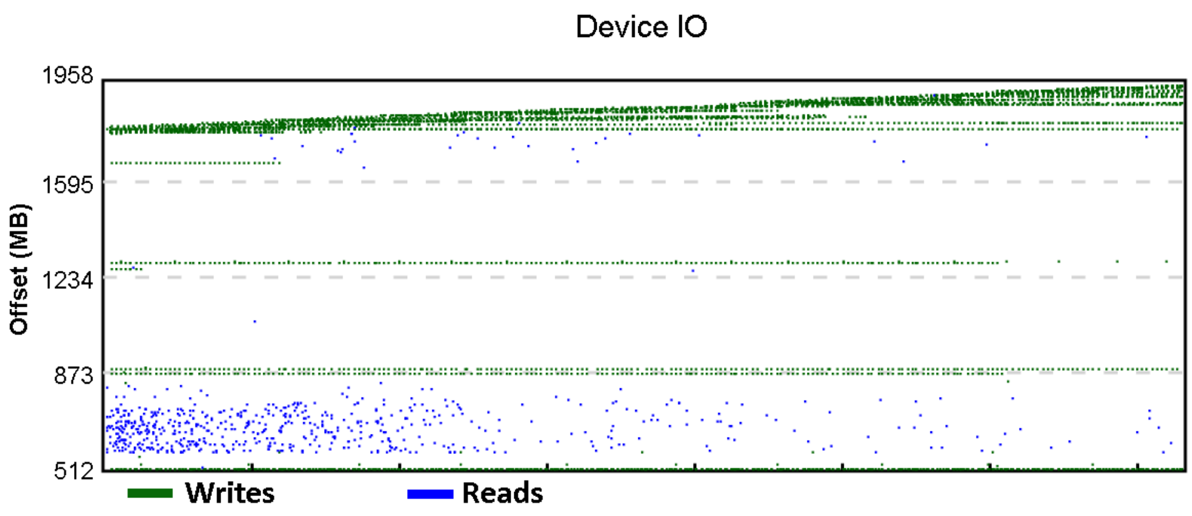


Figure 4.27: Blocktrace: SIAS-Chains - USB Flash Storage - 14 Warehouses - 2h. Small Dataset. Append operations are visible as monotonously increasing lines. Random read access in offset range [512, 873], sparsity indicates the effect of caching.

append characteristics of SIAS-Vectors (Figure 4.10) and the initial simulation (Figures 4.14 and 4.15).

The analog caching effect can be observed in the blocktrace diagram of SI, depicted in Figure 4.28. 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.

SIAS-Chains leverages parallelism by the selective read access using the scan over the VID_{map}. The *entrypoint* of each data item is fetched first.

The amount of write requests is significantly reduced. Table 4.2 shows the total amount of writes for SIAS-Chains and SI using *blkparse* on a large TPC-C dataset of 100 warehouses (WH). The reduction 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 4.5.5 for details). The one place invalidation of SIAS-Chains avoids the in-place update of visibility information. The trace diagrams (Figures 4.29 and 4.30) show that recently appended pages are accessed more often than pages that contain cold(er) tuple versions. Appends to each relation form horizontal swimlanes (green dots). In Figure 4.29 frequently read pages are visible as a horizontal blue line (marked as *New Data*). Pages that are accessed through the index is shown in Figure 4.29. This illustrates that under SIAS-Chains data is selectively read, forming a scattered distribution. These random reads utilize the ample I/O parallelism on Flash and represent no performance bottleneck.

SI in contrast shows the effect of the PostgreSQL background writer, which randomly writes pages at once, repeatedly. This effect is illustrated in Figure 4.30 that shows vertical almost equidistant write bars.

Write Reduction

The worst case for the SSD is that many different pages are updated in-place and only a small amount of data, e.g. invalidation information, is written. This occurs when only a sparse subset

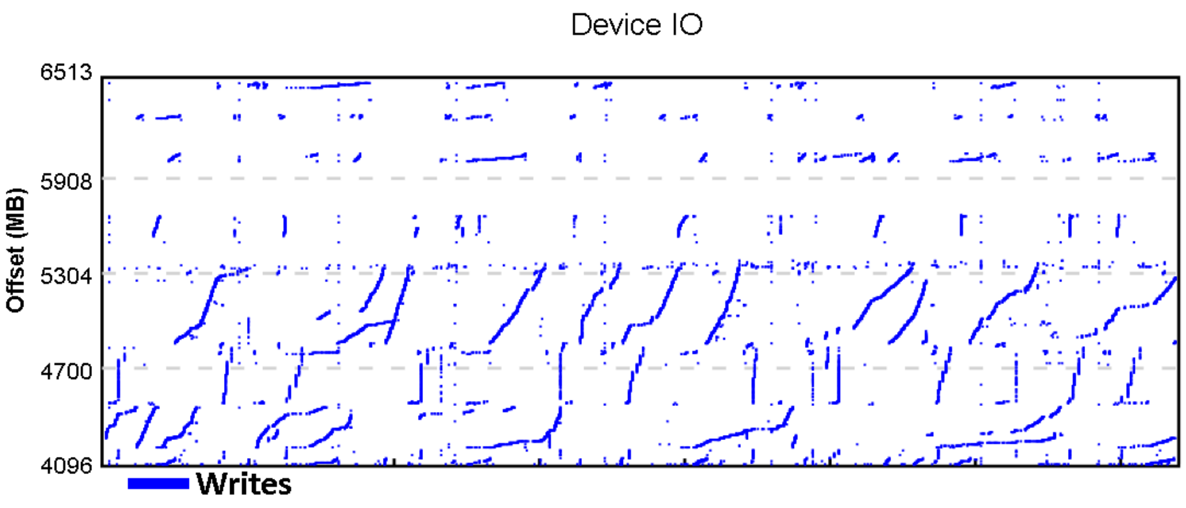


Figure 4.28: Blocktrace: SI - USB Flash Storage - 14 Warehouses - 2h. Small Dataset. Writes as *swimlanes* in increasing blocknumbers (HDD optimization).

of the relation is accessed and updated, e.g. price updates of different and distinct products in a warehouse that do not exceed a certain base-price.

In the evaluation we observe a *significant write reduction* (Section 4.5.5) by the SIAS-Chains algorithm, which is complementary to the SSD's inherent optimization mechanisms.

The SIAS concept reduces the *Visibility information* related write amplification that is caused by the version organization. Newly appended pages contain more *"net"* data.

We measured the amount of write reduction (issued writes) on a larger set of WHs. Since USB Flash was saturated by a relatively small amount of warehouses, we conducted these tests on the SSD. 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 (utilized) space. Table 4.2 depicts our findings. We found that SIAS-Chains significantly reduces the amount of write requests. In-place updates are transformed into appends.

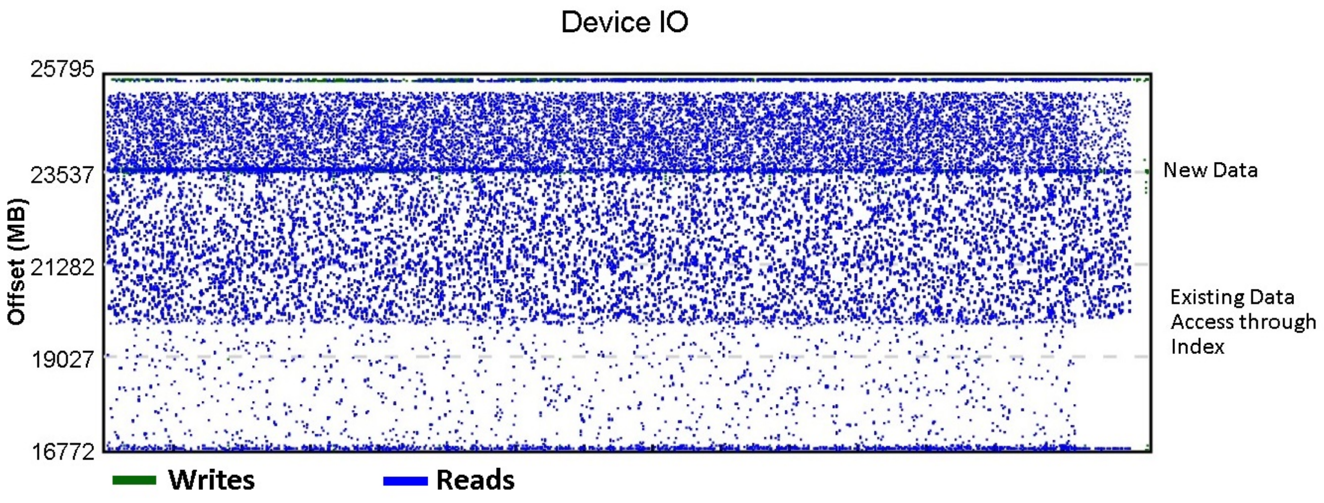


Figure 4.29: Blocktrace: SIAS-Chains - SSD - 100 Warehouses - 300 sec. Large dataset. Random read access. New data is appended to local append regions.

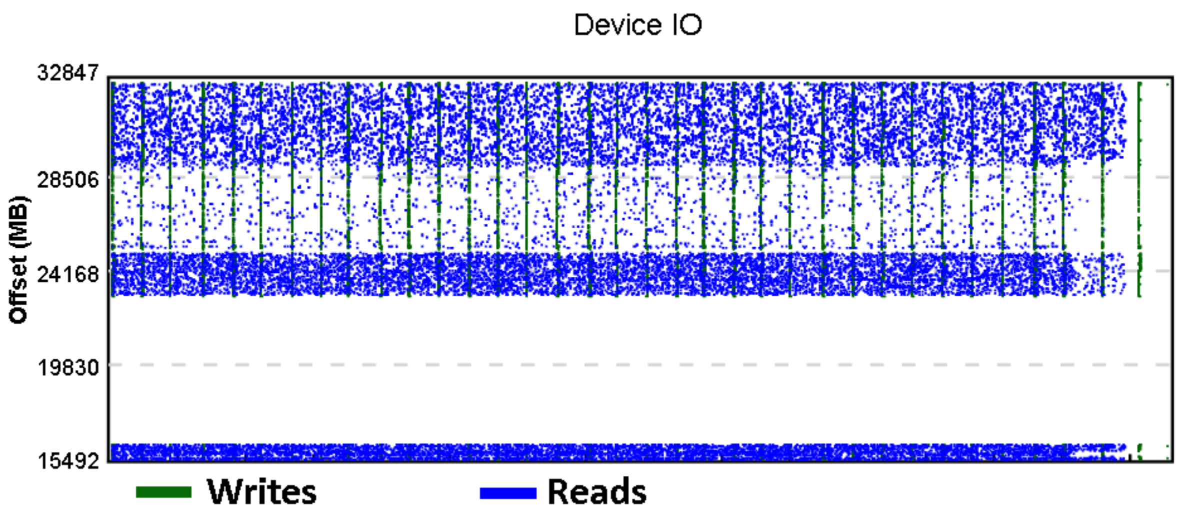


Figure 4.30: Blocktrace: SI - SSD - 100 Warehouses - 300 sec. Large dataset. In-place updates influenced by the background writer (intervals)

In the concept of SIAS-Chains a dense packing of tuple versions is created by appending only completely filled pages. The optimal space utilization and the highest write reduction is achieved by writing only completely filled pages. The amount of utilized space and issued writes is equal on such a setting (approx. 52 times - also reported in Section 6.4).

The amount of *write reduction* is dependent on the filling degree of each appended page. Nevertheless, the filling degree can be influenced by a threshold.

In order to investigate the efficiency of the append operation with default values, we configured SIAS-Chains with two different thresholds. Threshold $t1$ is the default setting of the PostgreSQL background writer process and $t2$ is defined by each checkpoint interval (piggy back).

Even when configured with the default thresholds, 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 4.2 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 the more conservative threshold $t1$ the reduction amounts to 65%. The optimal reduction amounts to approx. 55 times (Section 6.4). This is extremely beneficial for the Flash storage in terms of write stability, endurance and lifetime.

Tuple versions of different relations are not stored on the same page. Pages that belong to different relations are placed at different regions on the Flash memory (local append region). This avoids mixing tuple versions and pages and reduces contention, e.g. one relation receives more updates/inserts than another [35].

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 might physically lead to a whole erase-rewrite operation sequence, 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

Table 4.2: Write Amount (issued writes in MB) and Reduction (%) compared to SI. SIAS-Chains is configured with two different thresholds. SIAS-$t1$ is the default PostgreSQL background writer interval. SIAS-$t2$ piggy-backs the write operation with the interval of each checkpoint operation. In both configurations the pages are not filled completely, hence write reduction is threshold defined. The maximum amount of write reduction for TPC-C load is evaluated in Section 6.4.

Runtime(sec.)	**SI** (MB)	**SIAS-$t1$**(MB)	**SIAS-$t2$**(MB)	**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%

the new version, leaving the old one unchanged and updates the pointer in the in-memory VID_{map}.

The amount of *occupied space* also depends on the threshold, which directly influences the degree, to which each appended page is filled. The threshold defines when a new page is physically appended to stable storage. Threshold $t1$ is not suitable for the tuple LbSM, since pages are persisted too frequently and are only sparsely filled. This leads to suboptimal overall space utilization, although less net data is written (no overwrites). Appending too frequently leads to writing out pages that have a poor fill-factor, leading to wasted space and a higher amount of write requests. In SI the utilized space differs from the issued writes, since pages are updated in-place.

Threshold $t2$ delays the point of physically appending a page beyond that of threshold $t1$. 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 variations in runtime do not change the overall database size. This indicates that the pages under configuration $t1$ and $t2$ are not completely filled yet and the threshold can be further optimized. It is therefore possible to further increase the filling degree, resulting in a further write reduction and smaller database sizes.

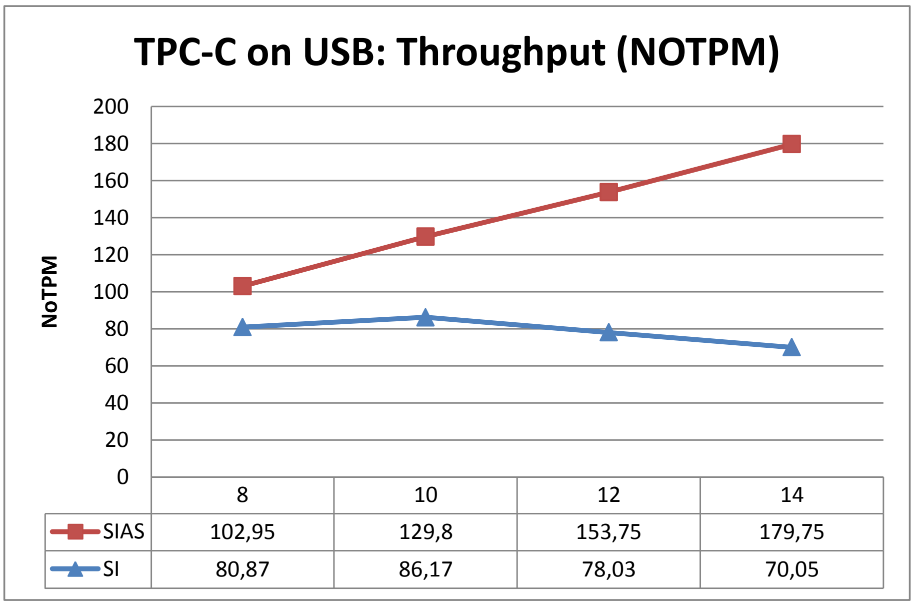
The optimal threshold for write efficiency is the maximum filling degree of a page. When configured with a filling degree of 100% the maximum write reduction of the SIAS tuple LbSM is approximately 52 times under TPC-C workload. This is evaluated in Section 6.4 and also reported in [35] and [38].

Performance on USB

We investigate the I/O performance behavior on the USB Flash storage. SIAS-Chains achieves stable, dependable write throughput and response times on Flash storage.

Throughput

SI manages to scale with up to ten WHs and subsequently reaches a plateau (response time approx. 5 sec.). At ten WHs SIAS-Chains yields 1.5 times the transactional throughput compared to SI and exhibits sub-second response times. The throughput of SIAS-Chains and SI is depicted in Figure 4.31. SIAS-Chains scales linearly and reaches the plateau at 18 WHs and max throughput at 20 WHs (Figure 4.39). While still providing a responsive system SI reaches peak



	8	10	12	14
SIAS	102,95	129,8	153,75	179,75
SI	80,87	86,17	78,03	70,05

Figure 4.31: SIAS-Chains vs. SI - Transactional Throughput. SIAS-Chains achieves higher throughput. It provides up to 2.6 times the transactional throughput.

performance with 86 NoTPM (New-Order Transactions Per Minute) and SIAS-Chains with 220, thus SIAS-Chains provides 2.6 times the transactional throughput.

Response Time

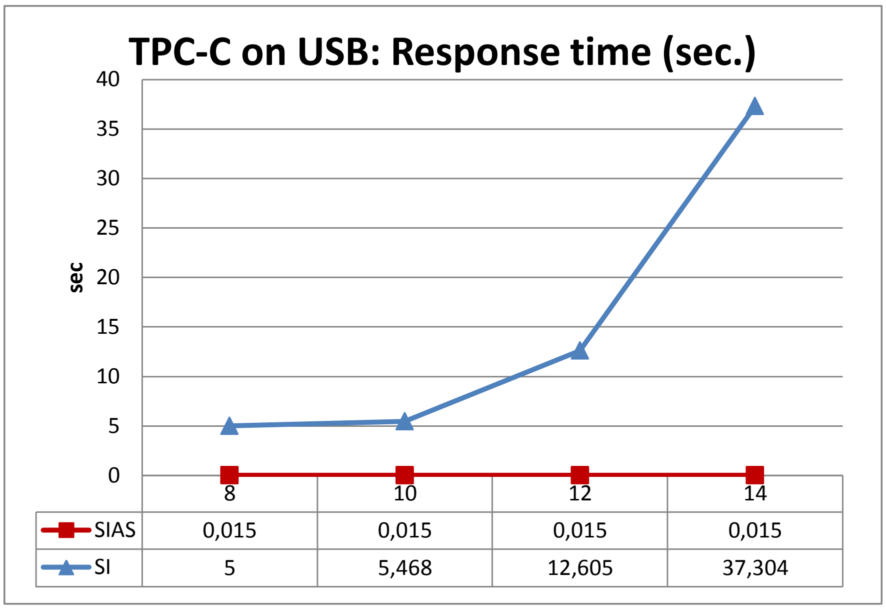
Figure 4.32 depicts the response time of SIAS-Chains and SI. SI exhibits response times below five seconds on scaling factors smaller than eight WHs. SI reaches the plateau at ten WHs and shows an exponential increase of the response times afterwards. SIAS-Chains exhibits stable sub-second response times up to a scaling factor of 16 WHs (blue curve Figure 4.32). SIAS-Chains shifts the kneepoint of the system and reaches the plateau at 18 WHs. This is 80% more tolerable load. At the time SI reaches the plateau with 5.468sec the response time of SIAS-Chains is 0.0015sec. Hence, SIAS-Chains further shifts the knee-point of the system and maintains a responsive system on peak loads.

Performance on SSD

We investigate the performance on our enterprise class Intel X25-E SLC Flash SSD. The DBT2 benchmark is instrumented with a varying amount of warehouses, 10 Clients and a runtime of two hours.

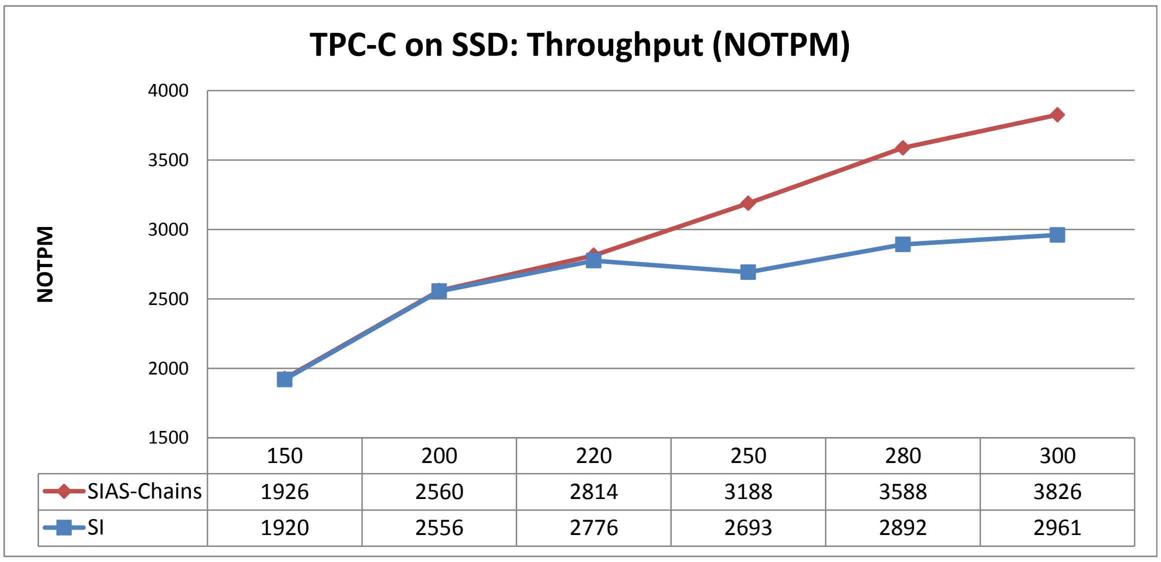
Throughput

The throughput, measured in NOTPM is depicted in Figure 4.33. SI reaches the plateau when the response time drops below five seconds at 220 WHs. After that point the system is no longer responsive and the graph of the throughput flattens. Benchmarking SI with more than 300 WHs resulted in a frozen system, hence no further performance comparison is reported above that point. SIAS-Chains scales linearly above that point, while still providing a responsive system. At 400 WHs SIAS-Chains delivers more than 5000 NOTPM, depicted in Figure 4.40.



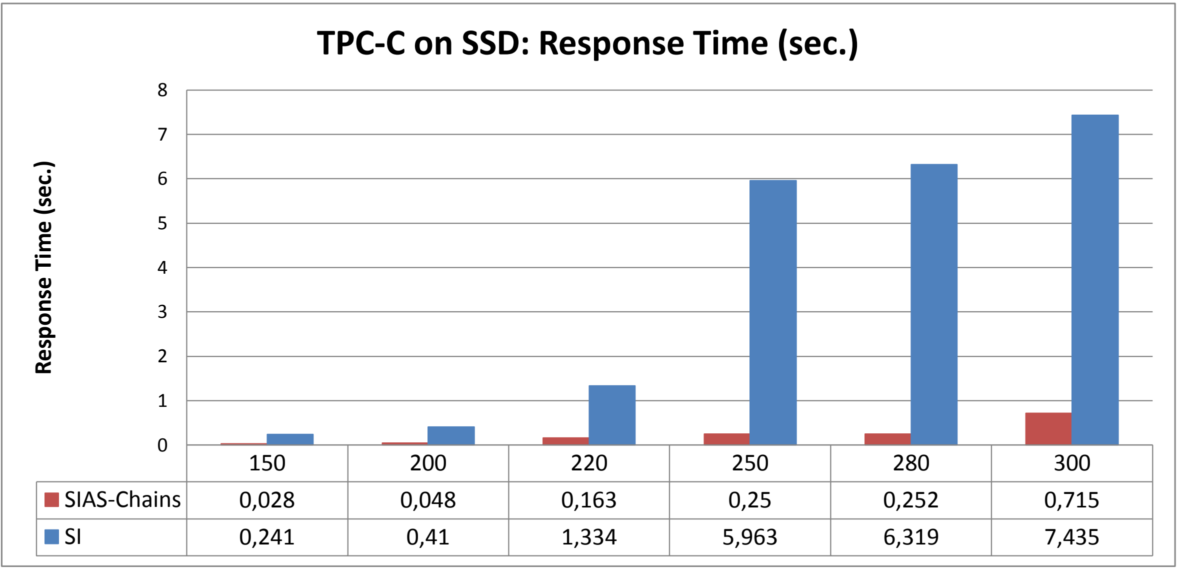
	8	10	12	14
SIAS	0,015	0,015	0,015	0,015
SI	5	5,468	12,605	37,304

Figure 4.32: SIAS-Chains vs. SI - Response Time. SIAS-Chains is shifting the kneepoint of the system, delivering a responsive system on peak loads.



	150	200	220	250	280	300
SIAS-Chains	1926	2560	2814	3188	3588	3826
SI	1920	2556	2776	2693	2892	2961

Figure 4.33: SIAS-Chains vs. SI - Transactional Throughput on SSD (ordinate begins with 1500 NOTPM). SIAS-Chains achieves higher throughput than SI.



	150	200	220	250	280	300
🟥 SIAS-Chains	0,028	0,048	0,163	0,25	0,252	0,715
🟦 SI	0,241	0,41	1,334	5,963	6,319	7,435

Figure 4.34: SIAS-Chains vs. SI - Response time on SSD. SIAS-Chains is shifting the kneepoint of the system, delivering a responsive system on peak loads.

Response Time

The response times are depicted in Figure 4.34. SI reaches the plateau after 220 WHs, the response time gets larger than five seconds. SIAS improves on SI by 13x lower response times at that point. SIAS delivers sub-second response times up to 300 WHs. The response time increases above that point and reaches 3.1 sec. at 400 WHs.

Performance on SSD RAID

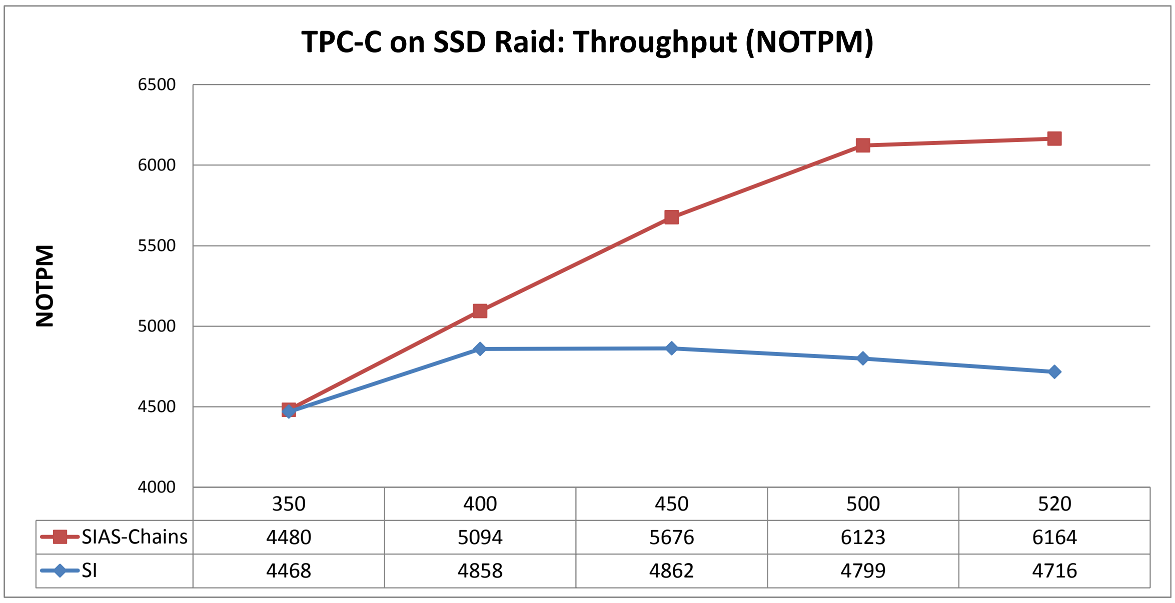
We benchmarked SIAS-Chains and SI on a RAID that comprises two Intel X25-E SSDs. The RAID is created using the *mdadm* software RAID in Linux. It is configured with RAID level 0 (stripe).

Throughput

SIAS-Chains achieves high throughput (Figure 4.35) with low response times (Figure 4.36) on SSD RAID storage. SIAS-Chains increases the throughput by up to 30%. SI reaches peak throughput with 450 WHs with a response time of 4.8 sec. providing 4862 NOTPM. SIAS-Chains reaches peak performance with 530 WHs with a response time of 3.3 sec. providing 6182 NOTPM (Figure 4.41).

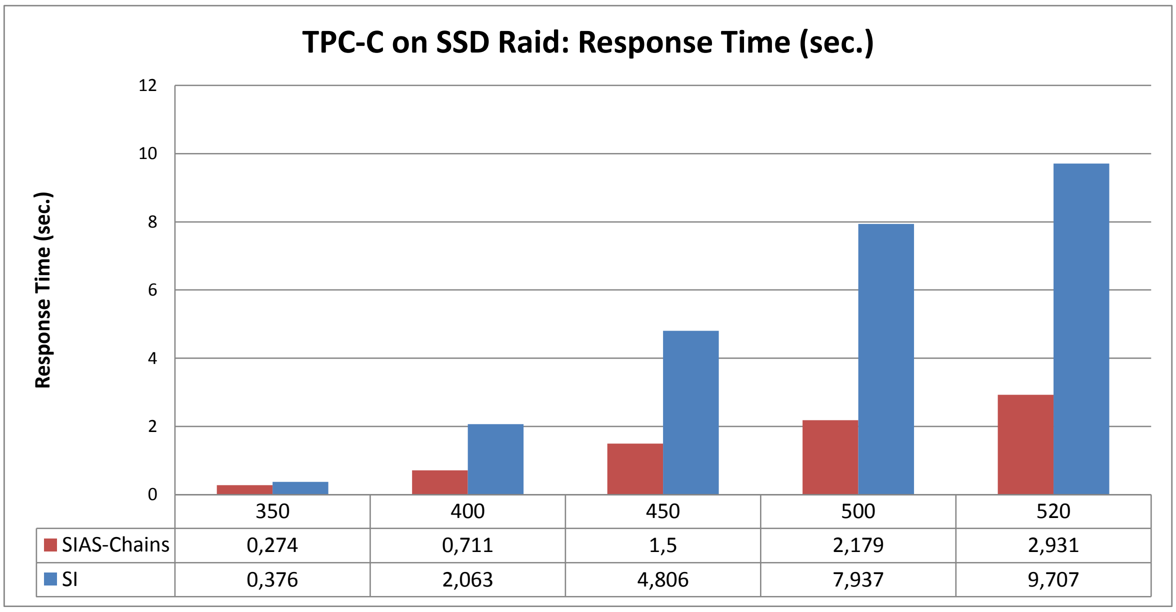
Response Time

Figure 4.36 depicts the response time of the system. SI features a response time below five seconds with up to 450 WHs. After that point the response time increases significantly up to the point where the system no longer responds to any query (above 520 WHs). SIAS-Chains stays responsive with more than 530 WHs where the response time is still below five seconds.



	350	400	450	500	520
■ SIAS-Chains	4480	5094	5676	6123	6164
◆ SI	4468	4858	4862	4799	4716

Figure 4.35: SIAS-Chains vs. SI - Transactional Throughput on SSD RAID (ordinate begins with 4000 NOTPM). Two Intel X25-E SSDs in software RAID (stripe).

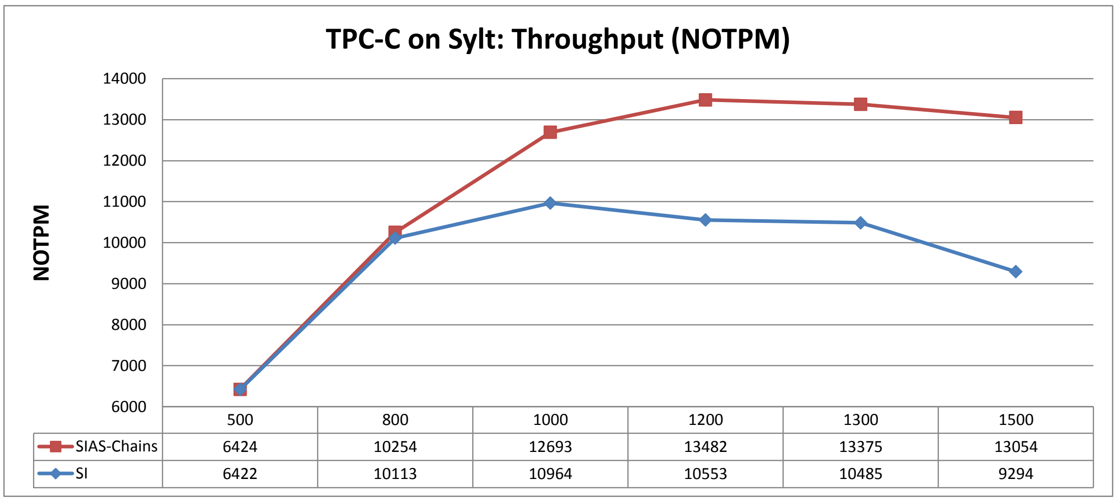


	350	400	450	500	520
🟥 SIAS-Chains	0,274	0,711	1,5	2,179	2,931
🟦 SI	0,376	2,063	4,806	7,937	9,707

Figure 4.36: SIAS-Chains vs. SI - Response time on SSD RAID with two Intel X25-E SSDs in software RAID (stripe). SIAS-Chains is shifting the kneepoint of the system, delivering a responsive system on peak loads.

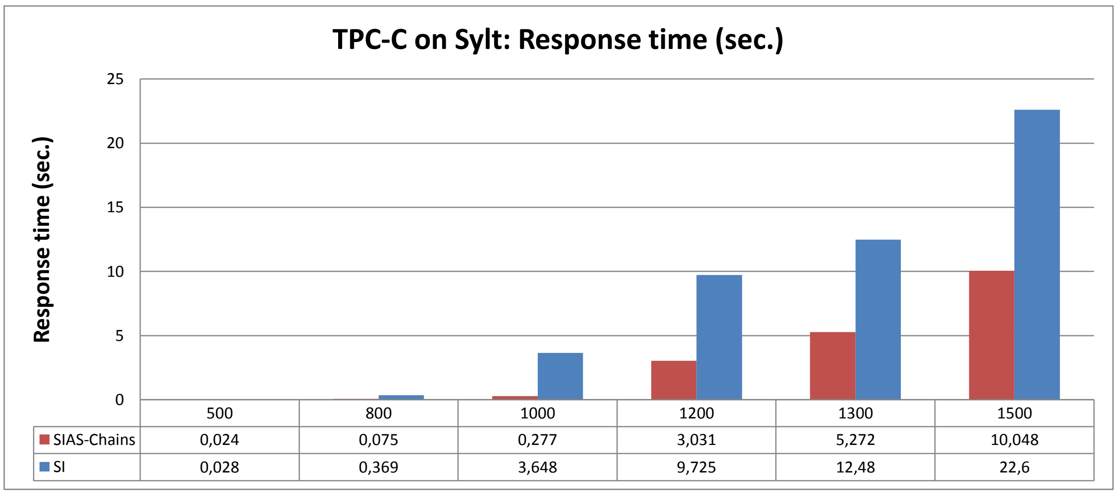
Performance on Sylt

Figure 4.37 depicts the throughput of SIAS-Chains and SI on the Sylt server. It is configured with six Intel X25-E SLC SSDs, 80GB of main memory and two Intel Xeon CPUs. This removes any bottleneck of computational power or available main memory.



	500	800	1000	1200	1300	1500
■ SIAS-Chains	6424	10254	12693	13482	13375	13054
◆ SI	6422	10113	10964	10553	10485	9294

Figure 4.37: SIAS-Chains vs. SI - Transactional Throughput on the Sylt server (ordinate begins with 6000 NOTPM). Six Intel X25-E SSDs in software RAID (stripe).



	500	800	1000	1200	1300	1500
■ SIAS-Chains	0,024	0,075	0,277	3,031	5,272	10,048
■ SI	0,028	0,369	3,648	9,725	12,48	22,6

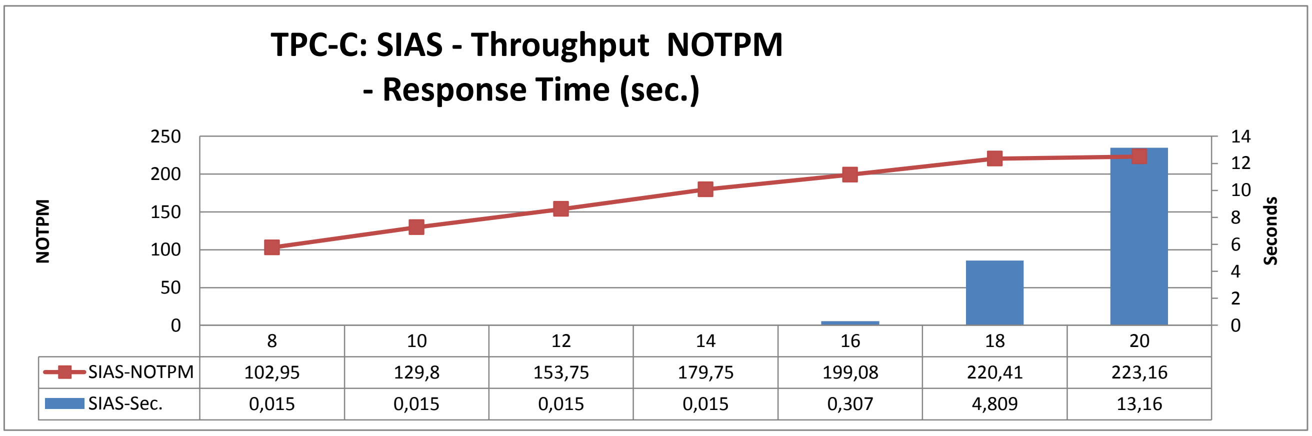
Figure 4.38: SIAS-Chains vs. SI - Response time in seconds on the Sylt server. Six Intel X25-E SSDs in software RAID (stripe).

Throughput

SIAS-Chains reaches the plateau after approx. 1200 WHs and SI reaches peak throughput at approx. 1000 WHs. After these points both algorithms decrease in throughput. The graphs in Figure 4.37 also show that the rate at which SI decreases is higher than that of SIAS-Chains.

Response Time

Figure 4.38 depicts the response time of SIAS-Chains and SI. As in the preceding benchmarks, the response time of SIAS-Chains is lower than that of SI on higher loads. On lower loads the response time is less or equal to that of SI.



	8	10	12	14	16	18	20
SIAS-NOTPM	102,95	129,8	153,75	179,75	199,08	220,41	223,16
SIAS-Sec.	0,015	0,015	0,015	0,015	0,307	4,809	13,16

Figure 4.39: SIAS-Chains response time (blue curve, right ordinate) and throughput (red curve, left ordinate) on USB. SIAS-Chains processes twice the load before reaching the plateau.

Scalability

The benchmarks show that SIAS-Chains, compared to SI, scales with a higher amount of warehouses while being capable of providing a responsive system with sub-second response times.

On USB SI scales only up to approx. 10 warehouses. After that point throughput and response time are impaired. The throughput of SI and SIAS-Chains on USB is depicted in Figure 4.31 and response times are contained in Figure 4.32, respectively.

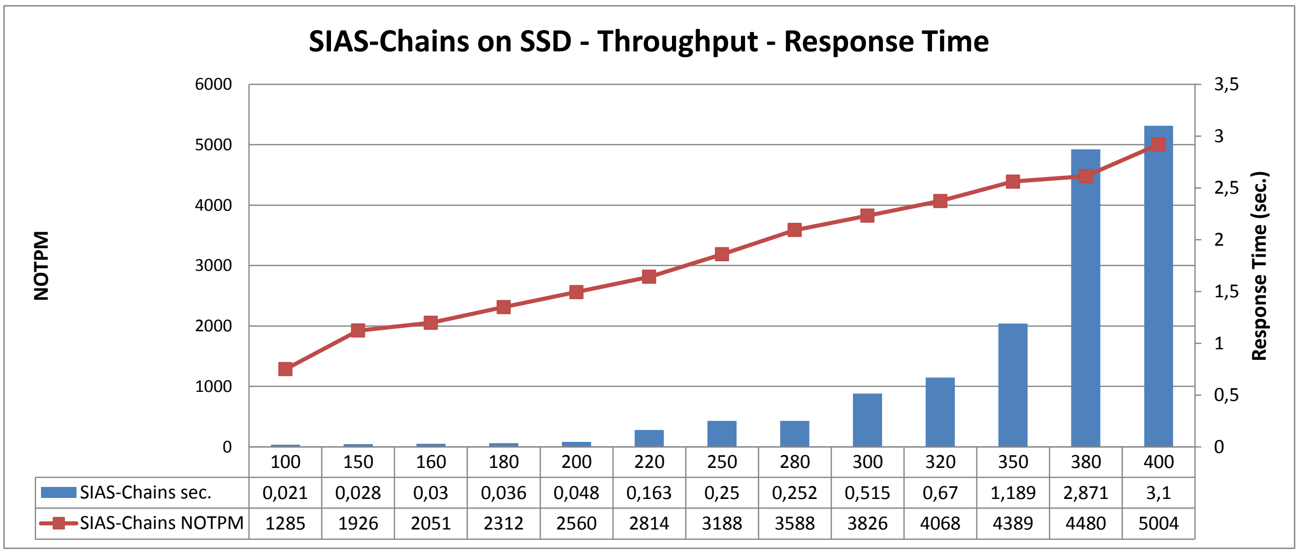
Since SIAS-Chains scales up to a higher amount of warehouses, the throughput and response times are depicted separately in Figure 4.39. The red curve in Figure 4.39 depicts the throughput and the blue curve depicts the response time of SIAS-Chains. With roughly double the amount of warehouses SIAS-Chains delivers a responsive system.

The benefits in response time are particularly impressive on SSD. SI reaches the kneepoint after 220 WHs, where SIAS-Chains still provides sub-second response times. SIAS-Chains scales up to 400 WHs with a response time far below five seconds.

This is due to the write reduction that is achieved by the SIAS-Chains algorithm: the positive effects of SIAS-Chains in terms of lower latency and higher transactional throughput can be attributed to the significant write reduction, caching of fresh tuple versions in pages and the lack of small in-place updates due to the new invalidation model.

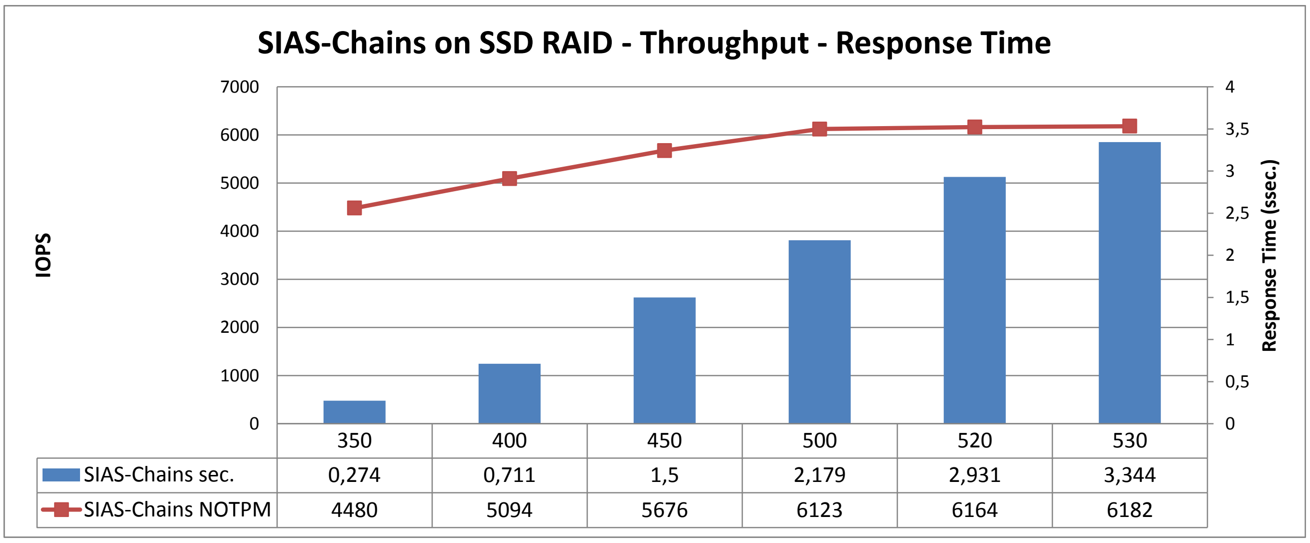
Performance on HDD

SIAS-Chains was developed for new storage media with the properties of Flash memory. It aims to optimize the I/O access patterns and explicitly leverages parallelism and read/write asymmetry. SIAS-Chains follows the idea that *reads are traded for writes and updates*. With SIAS-Chains we even observed clear advantages on HDD. We benchmarked SIAS-Chains and SI on a Seagate ST3320613AS S-ATA HDD with 7200rpm. On traditional symmetric storage writing cost equals reading cost and random read as well as random write access involves rotational and positioning delays.



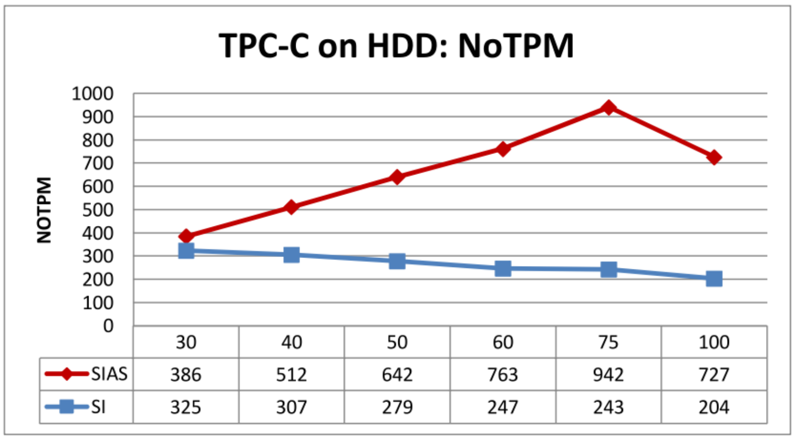
	100	150	160	180	200	220	250	280	300	320	350	380	400
🟦 SIAS-Chains sec.	0,021	0,028	0,03	0,036	0,048	0,163	0,25	0,252	0,515	0,67	1,189	2,871	3,1
🟥 SIAS-Chains NOTPM	1285	1926	2051	2312	2560	2814	3188	3588	3826	4068	4389	4480	5004

Figure 4.40: SIAS-Chains response time (blue, right ordinate) and throughput (red, left ordinate) on SSD (Intel X25-E). SIAS-Chains scales almost linearly up to 400 warehouses.



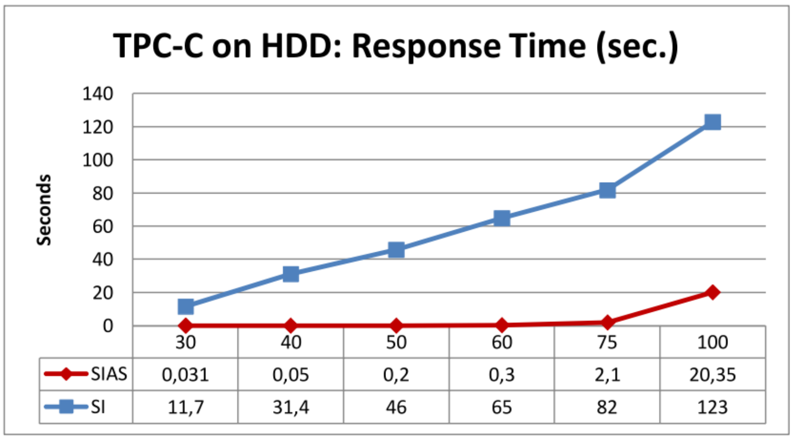
	350	400	450	500	520	530
🟦 SIAS-Chains sec.	0,274	0,711	1,5	2,179	2,931	3,344
🟥 SIAS-Chains NOTPM	4480	5094	5676	6123	6164	6182

Figure 4.41: SIAS-Chains response time (blue, right ordinate) and throughput (red, left ordinate) on SSD RAID (2x Intel X25-E). SIAS-Chains scales almost linearly up to 530 warehouses.



	30	40	50	60	75	100
SIAS	386	512	642	763	942	727
SI	325	307	279	247	243	204

Figure 4.42: SIAS-Chains vs. SI - Throughput. SIAS-Chains optimizes for asymmetric Flash storage, yet traditional symmetric storage also benefits from the approach. Throughput is improved.



	30	40	50	60	75	100
SIAS	0,031	0,05	0,2	0,3	2,1	20,35
SI	11,7	31,4	46	65	82	123

Figure 4.43: SIAS-Chains vs. SI - Response Time. SIAS-Chains optimizes for asymmetric Flash storage, yet traditional symmetric storage also benefits from the approach. Average response times are lower on SIAS-Chains due to the write retention and grouping of new inserted tuple versions.

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 the SIAS-Chains approach 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.

On HDD SIAS-Chains improves on SI in terms of transactional throughput and especially on response time. The lower response time indicates the benefit of the co-location of fresh tuple versions. New appended pages contain fresh data and it is more likely that on a subsequent access a page is still cached. SI co-locates tuple versions of a data item, supporting a clustering of data. SI exhibits high response times. The system stays responsive below 30 WHs. SIAS-Chains provides a responsive system with up to 75 WHs. It can be observed that on a scaling factor of 100 WHs the performance of SIAS-Chains suddenly drops, suffering a 10 times increase in response time compared to 75 WHs. At this point random reads dominate, since SIAS-Chains

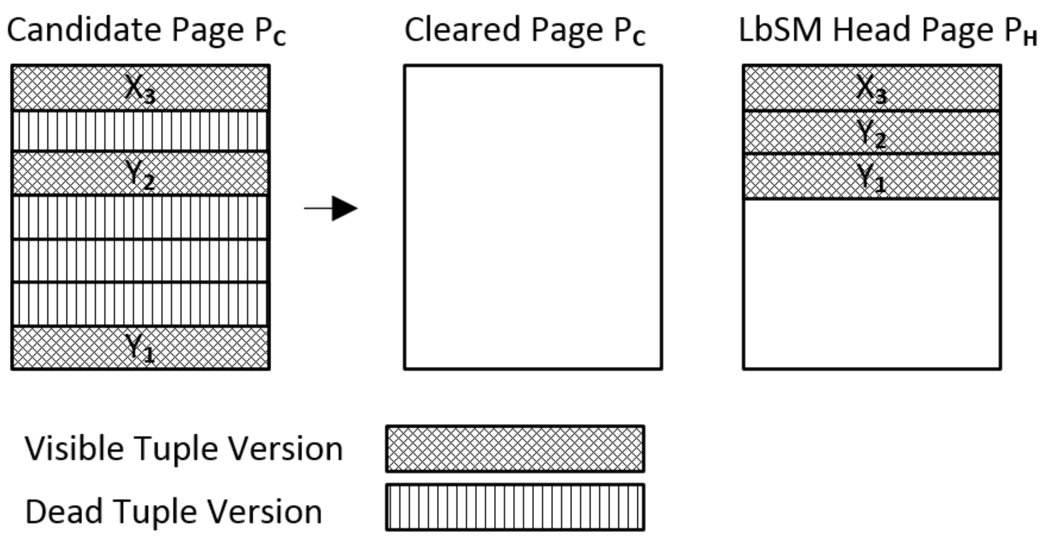


Figure 4.44: Garbage Collection concept. First a candidate page P_c is chosen. Valid tuple versions X_3, Y_2, Y_1 are re-inserted (re-located) in the append storage, here at the LbSM head page P_H. The mapping structure VID_{map} is updated and P_c is erased, reclaiming the occupied space.

does not maintain a clustering, a property inherited by the append based storage management. It tends to produce more random reads, hence the scaling factor of 100 warehouses shows a sudden decrease in throughput and increase in response time.

4.6 SIAS Garbage Collection

The basic concept in MV-DBMSs to reclaim space on the append storage is using a garbage collection (GC) mechanism which: (i) finds a candidate (victim-) page P_c that is chosen to be garbage collected, (ii) re-inserts still valid tuple versions and (iii) discards dead tuple versions of that page. Currently the SIAS-Chains prototype does not perform space reclamation techniques. All three steps offer interesting points of research, which are currently under investigation in our group and left for future work. Figure 4.44 depicts the basic concept of the GC.

Obviously it is more beneficial to chose a candidate page that contains many dead tuples. An unsuitable candidate page comprises a high fraction of visible tuple versions.

Granularity

Flash storage can only be erased in erase unit granularity, hence the smallest unit for GC can be defined by the granularity of one single erase unit.

GC triggered by the DBMS is a deterministic process 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. Performance (latency) outliers that are caused by indeterministic execution of the on-device GC can be avoided.

Data Placement

Rather than re-insert valid tuple versions they can be efficiently re-located. Hot and cold tuple versions can be grouped, sorted and stored appropriately. It can be beneficial to group tuple versions which are updated frequently, such that a larger potion can be garbage collected.

The SIAS-Chains concept of using VIDs complements the re-insertion/ re-location of visible tuple versions. When a tuple version receives a different TID during the garbage collection process, the VID_{map} gets updated and the VID points to the new position of that version. Existing index entries do not have to be changed on a re-location of a tuple version, since they store the VID instead of the TID.

This transfers yet more control over the Flash storage into the MV-DBMS and complements the NoFTL approach [46].

Maintaining the Version Chain

If a visible version is found which is not the entrypoint of a data item, the whole chain is re-inserted/ re-located. During that operation, the placement of the entrypoint and the still visible predecessor tuple version(s) can be reconsidered. For instance, all entrypoints of the data items can be stored on a different page than their corresponding predecessors, which is beneficial under the assumption that the predecessor tuple versions will become dead tuple versions *soon* (the oldest running transaction can only access a newer version) and can be discarded on a subsequent GC call.

Indexing

A serious consideration is the treatment of indexed data items. When an indexed data item is relocated during the execution of the GC. In a traditional MV-DBMS all indexes have to be updated in order to point to the new position of the data item (its tuple version). In SIAS the VID_{map} keeps track of the entrypoint and the index structures redirect accesses to the VID_{map}, hence only the VID_{map} has to be updated. The index structures do not have to be updated since they already store the VID. This property complements a simplified GC. The indexing mechanism in SIAS-Chains is explained in Chapter 5.

4.7 SIAS Buffer Management

The SIAS buffer manager is designed to be incorporated with the other components of the SIAS approach. The SIAS concept employs a simplified buffer management that performs a *write retention* of buffered pages which are about to be appended to the LbSM.

Pages are appended to the LbSM after they are completely filled. Once written, such pages are immutable, apart from garbage collection and space reclamation.

A configurable parameter defines the amount of buffered pages per relation. Appending to only one page creates a bottleneck since concurrently executed transactions have to share the buffered page for modifications.

The SIAS-Chains prototype was configured with two different time-based thresholds to write out buffered pages. This led to the event that pages have been physically appended to the LbSM although they are not completely filled, thus having negative impact on overall space consumption and the amount of issued host writes.

4.8 SIAS Recovery

SIAS-Chains and SIAS-Vectors do not impinge on the MV-DBMS's inherent recovery mechanisms. The write ahead log (WAL) as well as the MV-DBMS's inherent mechanisms for recovery are not impaired. The buffer manager performs a threshold configured write retention which delays the time until a single page is allowed to be physically appended and written out to the storage media, yet WAL is not affected.

The SIAS-Chains VID_{map} is persisted in a container relation on regular shutdown of the database. Nevertheless, as all information that is needed for a reconstruction after a crash is stored on each tuple version (Section 4.5.2), no additional effort is taken to recover the VID_{map}.

The VID_{map} can be recovered after a crash by a *piggybacked* scan of each relation. Since each tuple version stores it's VID, the VID_{map} can be successively re-built. The same is true for the SIAS-Vectors datastructures. The bitmap vectors can be re-built using the on-tuple information.

4.9 SIAS Discussion - Worst Case Scenario

The search for a worst case scenario for the SIAS concept (based on the SIAS-Chains implementation) is an interesting discussion.

Maximum Overhead of the data structures

In order to experience the maximum overhead of the data structures, there should be relatively short version chains. For instance if only one tuple version per data item exists, then the overhead of keeping the VID_{map} is maximized. Yet this is an unrealistic scenario, since the MV-DBMS manages multiple versions of data items - which is one of the basic assumptions of multi-versioning.

Traversal of the Version Chain

On a traversal of the version chain a predecessor tuple version is fetched. In a worst case scenario, many predecessor tuple versions have to be fetched in order find the visible one. This can occur on mixed workload when a long running transaction needs to access an older tuple version of a data item. Yet this only applies to an old transaction. Newly started transactions are not affected, since those have to read the newest committed tuple version of a data item. In an OLTP system it is most likely that the visible tuple version is denoted by the entrypoint or its immediate predecessor, as our statistics on the visible part of a version chain show (Section 4.4.3, Figure 4.20).

5 Access Methods - Searching and Indexing

Contents

5.1 SIAS-Chains: Scan - Introduction 89

5.1.1 SIAS-Chains - VID_{map} Scan 89

5.1.2 SIAS-Chains - Full Relational Scan 90

5.2 SIAS-Chains: Indexing - Introduction 91

5.2.1 SIAS-Chains - Indexing 91

Data stored on Flash is accessed by a scan or by using an index. The index access enables selective point queries which usually result in small random reads, yet also sequential access is possible on range queries on an index.

5.1 SIAS-Chains: Scan - Introduction

SIAS-Chains enables two variants of scan operations. The SIAS-Chains VID_{map} scan and the traditional relational scan. In the following both scan variants are discussed.

5.1.1 SIAS-Chains - VID_{map} Scan

This method performs a scan on the VID_{map}, fetching the entrypoints of all data items first. The VIDs in the mapping structure are accessed in ascending or descending order. This access pattern is more likely to create random reads on the flash storage, since the entrypoints are not clustered. Pages that do not contain an entrypoint or any visible tuple version are not fetched. A page needs to contain at least one visible tuple version in order to be fetched from the storage. Tuple versions are checked for visibility beginning with the entrypoint - as soon as the visible version of the data item is found, no older versions need to be fetched and checked.

Algorithm 5 describes the scan using the VID_{map}. The VID_{map} is accessed first to determine visible tuple versions. This variant is implemented in the SIAS-Chains PostgreSQL prototype. *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 cirteria. The commit timestamp is not stored on the tuple version and therefore cannot be checked directly. 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 5, 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". SIAS-Chains scans the VID_{map} first and enables more selective I/O as shown in the evaluation (Section 4.5.5).

Algorithm 5 SIAS-Chains Scan over VID_{map}

```
1: procedure SCAN($Transaction\ tx$)
2:     For each VID v $\in VID_{map}$ {
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 ($X_{v.create} \leq tx_{id}$)&& ($X_{v.create} \notin tx_{concurrent}$)
20:    }
21: end procedure
```

5.1.2 SIAS-Chains - Full Relational Scan

The traditional implementation of relational scan provided by PostgreSQL, 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. This access pattern fetches all pages from the flash storage (ascending or descending page number), even if they do not contain any visible tuple version. Since SSDs enable fast random reads, the traditional scan is inefficient, since each tuple version has to be checked. In addition, this scan uses the logical block numbers which are likely to be physically remapped to different positions on the SSD (black-box abstraction). On the traditional scan the same visibility criteria as in Algorithm 5 are applied. Such a relation scan first fetches all the tuple versions and sorts them for matching attributes corresponding to the query. Subsequently each tuple version has to be checked for visibility individually. Hence a tuple version becomes a visible *candidate*.

SIAS-Chains executes the same base algorithm for checking a tuple version: the entrypoint of the data item is fetched and the visible version (if it exists) is determined as in Algorithm 5 - 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 VID_{map} which uses only the entrypoints, since all potential versions of a data item need to be checked thus involving additional memory consumption and CPU cost. The advantage of the sequential scan on traditional HDDs does not hold for SSDs. Instead, the individual access to the entrypoints exploits the parallelism of the SSD.

In the absence of the VID, the SIAS-Vectors approach relied on the traditional scan, realized by a memcopy of the vector datastructes for each scan. The copied vector data structure is used to identify entrypoints that are valid at the snapshot of the scan. Concurrently updated and inserted tuple versions are shielded from the snapshot. Details on the scan of SIAS-Vectors are discussed with the SIAS-Vectors visibility check, described in Section 4.4.2 (Algorithm 2).

5.2 SIAS-Chains: Indexing - Introduction

SIAS-Vectors utilizes bitmap vectors in order to identify the entrypoint of each data item. This approach is suboptimal for index access: although the entrypoint can be determined very fast by a simple bit comparison, it lacks the functionality of finding a specific entrypoint to a given tuple version that is not an entrypoint. Yet that kind of access occurs on the access path using an index.

SIAS Chains improves on SIAS-Vectors. Since the bitmap vectors are no longer necessary and the information about specific entrypoints is coded within a chain of tuple versions that belong to a data item. The entrypoint of each data item is stored within the SIAS-Chains mapping structure. Using this paradigm where all versions of a data item can are identified by a virtual ID (VID), it is possible to use that VID within an arbitrary index structure. Instead of the tuple version's ID that stores the physical position of (TID, row ID, lists etc...) the VID is stored. This is implemented in the SIAS-Chains implementation's tree index structure (B^+tree). There are no additional requirements on the type/layout of the search key (key). It is simple adaptation of the value that is stored for a given search key. E.g. in traditional B^+Tree index structures there are three different record formats and their corresponding counterparts in SIAS-Chains.

- $< key, TID > -> < key, VID >$.: the direct pointer to the location where the tuple is stored, which becomes an index to the VID structure and points to the data item's entrypoint.
- $< key, Tuple > -> < key, DataItem >$: a unique data item. Note: there is no difference, since it is assumed for this case that the data item stays unchanged.
- $< key, \{listofTIDs\} > -> < key, \{listofVIDs\} >$: a list of TIDs which becomes a list of VIDs.

When a tuple version matches a search key the access is made to the VID_{map} leading to the entrypoint of the data item. This also entails an important property: In case that an attribute of a data item is updated, where the MV-DBMS creates a new version of the data item, the (VID) entry within the index can stay the same if the search-key attribute has not been changed. The visible tuple version is then determined by the SIAS-Chains algorithm (Section 5.1).

5.2.1 SIAS-Chains - Indexing

Any index structure can be adapted at the cost of an additional indirection. 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

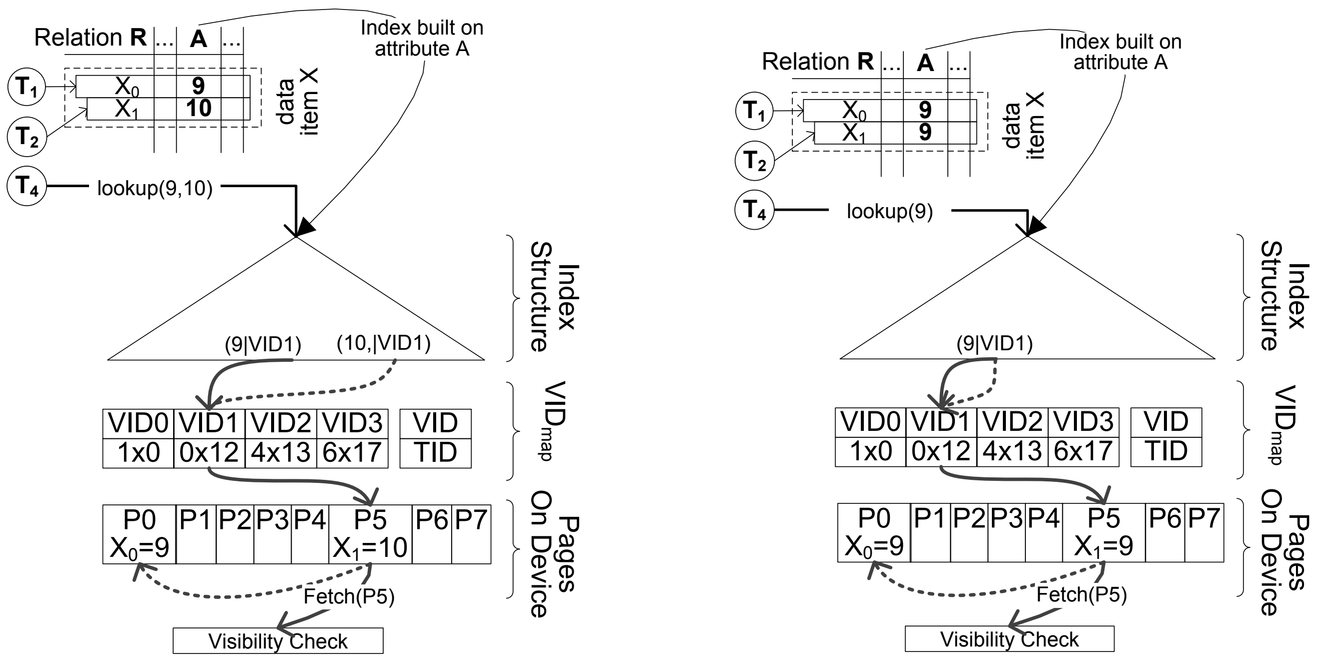


Figure 5.1: SIAS-Chains Indexing: Index Key Update (left). SIAS-Chains Indexing: Non-Index Key Update (right). Access is served by the VID_{map} and forwarded to the entrypoint of the data item.

there exists exactly one VID_{map}. SIAS-Chains uses the very same VID_{map} for all access paths. Figure 5.1 depicts the indexing in SIAS-Chains. In both figures 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 (appended on $P0$). T_2 updates X, changed the value to 10 and creates X_1 (appended on $P5$). 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.

Example 2: Update on a non-indexed attribute

Figure 5.1 (right) depicts the update to X 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 5.1 right, 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.

6 Storage Management

Contents

6.1 Append Storage - Introduction 93
6.2 Append Granularity 94
6.2.1 Page Append LbSM 96
6.2.2 Tuple Append LbSM 96
6.3 Data Placement and Layout 96
6.3.1 Global Region 97
6.3.2 Local Region 97
6.4 LbSM Evaluation 98
6.4.1 Simulation. 99
6.4.2 Validation 101
6.4.3 Results 103
6.5 Read Optimizations 114
6.5.1 Index-Snippet. 115
6.5.2 Ordered/Unordered Append Log. 118

Log-/Append-based Storage Managers (LbSM) are a forthcoming trend in modern MV-DBMS. Their principle of implementing each logical modification of data as a physical append operation complements the properties of Flash memories. This Section examines existing approaches to LbSM and proposes, along with SIAS, a new approach to LbSM that is integrated into a MV-DBMS.

6.1 Append Storage - Introduction

The concept of LbSM is well known from file systems (Rosenblum and Oosterhout [97]). LbSM append data portions to the logical end of log-organised storage, thus avoiding physical in-place updates and random writes. Each logical modification operation (update, insertion, deletion) is physically realised as an append operation. In LbSM a once appended page is immutable. Random writes, which are suboptimal for Flash storage, are transformed into appends. Appends are more suitable for Flash memories, nevertheless follow up read access is more complex. Since multiple versions of the appended data (page) may exist, the correct version of the data has to be identified. This entails garbage collection and more complex space management. Outdated (invalidated) data can be removed in order to free occupied space. LbSM as traditionally proposed are not natively integrated into the MV-DBMS. They do not address issues related to version organisation caused by, e.g., the in-place invalidation of outdated tuple versions (write overhead/ amplification).

The type of versioning directly influences the amount of mapping and write overhead the LbSM needs to manage. The versioning paradigm of MV-DBMS entails the concept of out-of-place updates. Although this alleviates the append of new data in principle, it is not applied in traditional approaches which address special properties of symmetric storage (rotational delay, positioning time). Current MV-DBMS maintain a clustering (primary index, most frequently read data, etc...) by performing in-place updates to optimise for sequential read access. They write data to arbitrary (random) positions, update data and invalidate old versions in-place. This effort effectively creates contraproductive write patterns for the LbSM and leads to additional write amplification, since each update of an already appended page needs to append a new version of that page.

This renders the traditional assembly, where the LbSM is realized as a seperate layer, of a MV-DBMS suboptimal. Furthermore: on the one hand the MV-DBMS maintains a clustering of logical pages and on the other hand the LbSM creates a remapping of those pages to ascending physical block addresses (PBA) on the storage media which destroys the physical clustering. New and/or updated data can not be clustered with already appended data. Approaches using delta stores (such as [81]) still require relatively expensive merge operations in order to maintain the clustering and induce additional overhead on read access.

As traditional LbSM approaches primarily address the high throughput of large sequential writes on HDDs, they have mostly been used in write heavy environments. Modern SSDs require optimized access methods that leverage low latency and fast random reads. Access methods need to address (relatively) low random write I/O, fast sequential writes and the vulnerability to wear.

We claim that the integration of the LbSM into the MV-DBMS effectively addresses the characteristics of modern storage technology. The LbSM needs to be implemented and integrated within the architecture and algorithms of modern MV-DBMSs. As described in Chapter 3 the SIAS approach combines version visibility and transactional management with the LbSM.

Two important factors of LbSM have been identified and investigated in this work (published in [35]):

- The *size* (granularity) of the logical and physical append unit - Section 6.2.
- The *data placement and layout* of the physical append operation (region of append) - Section 6.3.

In the following these two factors are further investigated and evaluated based on [35], showing that a *tuple LbSM* with *local append regions* performs best under OLTP workload.

6.2 Append Granularity

The device interface to the Flash SSD (inherited from the interface of traditional storage) implements a block device and commercially available SSDs are blockoriented. Therefore, the smallest *physical* append unit is constrained by the block size, nevertheless the logical append unit can be a fraction or a multiple of that size.

The granularity of the logical append unit is a critical component, on the one hand appending in page granularity (page mapping) is convenient since the LbSM can be implemented as an additional layer *outside* of the MV-DBMS and several components of the MV-DBMS can stay unaltered. Such traditional page mapping LbSM approaches utilize coarse grain granularity appends and implement a mapping of logical to physical blocks.

On the other hand page mapping introduces threefold overhead:

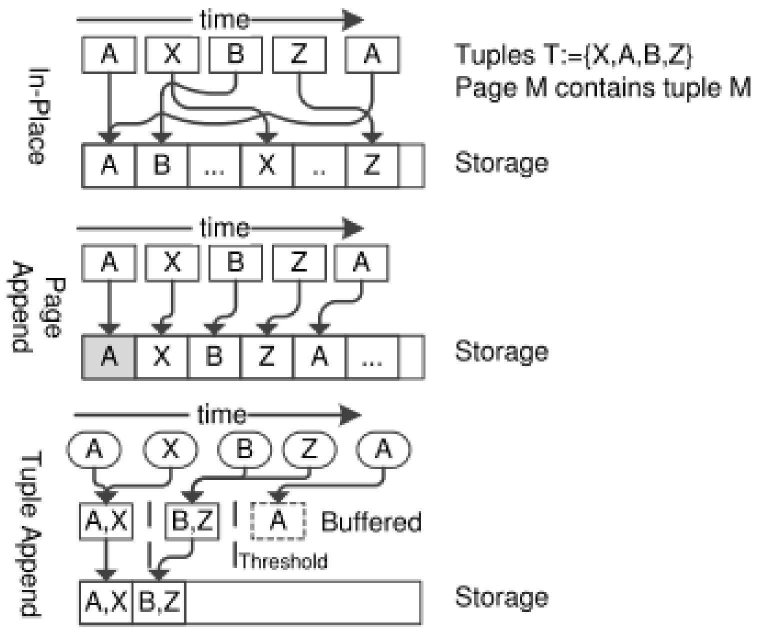


Figure 6.1: In-Place Storage Managers update pages in-place. Page LbSM append whole pages in the order in which they arrive. Tuple LbSM require a write retention mechanism that buffers tuple versions before they get physically appended. A threshold is required (time, work or space defined).

- *Mapping overhead*.

  Mapping overhead is induced by maintaining a mapping table of logical block addresses to physical block addresses.

- *Version handling*.

  Additional version handling of appended pages: updates of already written pages render them invalid and new versions of such pages are appended. Garbage collecion of invalidated pages is necessary to free space.

- *Write amplification*.

  Write amplification results from MV-DBMS version management. E.g., the update of a small unit (data item) within a page where the whole page has to be (re-)written (remapped and appended). Each logical modification, especially in-place invalidation in MV-DBMS, leads to the rewrite of a whole page.

The choice of the size of the mapping table in traditional LbSM is a tradeof: large append units reduce the size of the mapping table but increase the write amplification on small modifications, small append units reduce the write amplification but increase the size of the mapping table.

In contrast, the integration of the LbSM into the MV-DBMS renders the additional layer obsolete, alleviating the mapping table and version management overhead. A coupling of the LbSM functionality to the MV-DBMS transactional model, including the invalidation model, enables appending in smaller logical units (single tuple versions).

Furthermore, the knowledge about the data on the transactional level improves the quality of the append operation. Other optimizations can be applied, for example, appending sorted runs (Section 6.5).

The SIAS approach (Section 4.3) uses a tuple version as the logical append unit and the (much larger) page size as the physical append unit which is transferred to the storage device.

6.2.1 Page Append LbSM

Traditional approaches to LbSM(page LbSM) append in logical granularities of pages (or multiples of them). They implement an additional layer between the MV-DBMS and the storage media. Unaware of the content of each appended unit, they maintain a mapping of a logical block address to the corresponding physical block address. On an update, a logical page is remapped and a new physical block address at the head of the append-log is assigned to that page. Figure 6.1 illustrates the page append operation. Page A is updated, a new version of that page is subsequently stored at the end of the LbSM. This creates out-of-place updates and effectively addresses the in-place update issue on Flash [109]. Nevertheless, since the logical granulariy is defined by the page/block size, even small updates to a page lead to the update of the physical page, thus creating write amplification. For example, a whole page needs to be remapped, independent of the amount of changes to that page, even if only a single data item is updated. If the MV-DBMS is not aware of the page LbSM and Flash storage underneath, it issues in-place invalidations which cause small updates to pages that have to be re-inserted and remapped.

6.2.2 Tuple Append LbSM

The SIAS approach is a tuple append LbSM (tuple LbSM) approach. Each modification of a data item leads to the creation of a new tuple version of that item which is logically appended to a new page. Small updates do not lead to the remapping of a whole page, instead new tuple versions are created while old tuple versions stay unchanged. Small data portions get collected and are inserted/appended to a new page, which is written by the LbSM when it is completely filled or once a threshold (time, amount of committed transactions etc...) is reached. The process is depicted in Figure 6.1 where data item A is updated and appended to a new page. The page holding the old tuple version of data item A is left unchanged. The new allocated page is held in the buffer until a threshold is reached. The LbSM functionality is migrated into the MV-DBMS and no additional external mapping structures have to be employed, since the MV-DBMS already maintains an explicit mapping. This integration requires a change of the invalidation model within the transactional model of the MV-DBMS and a new paradigm to version invalidation, as described in Section 4.3.

6.3 Data Placement and Layout

The *physical region* of the append is an important factor which directly influences the I/O pattern. Whether data-pages of one relation is mixed with data of another relation, or if data of different relations is physically seperated. This is evaluated in this work using multiple regions, e.g., one per relation which results in multiple append write streams, or using one global append region.

Each individual page contains tuple versions of exactly one relation.

6.3.1 Global Region

A LbSM with one *global region* which is the same for all relations appends pages that belong to different relations to a single region (Figure 6.2). Thus using one writer process for all relations that creates a mixture of pages that belong to different relations. Pages are appended in the order in which they arrive at the LbSM. Data pages of different relations are appended to the same writer, hence the amount of written data is forming a single write stream.

6.3.2 Local Region

A LbSM with *local regions* (also depicted in Figure 6.2) appends each page of a single relation to a separate region, thus using multiple writer streams - at least one for each relation. Pages that belong to different relations are not mixed within a region. This also influences space management, since the append operation is executed individually for each relation.

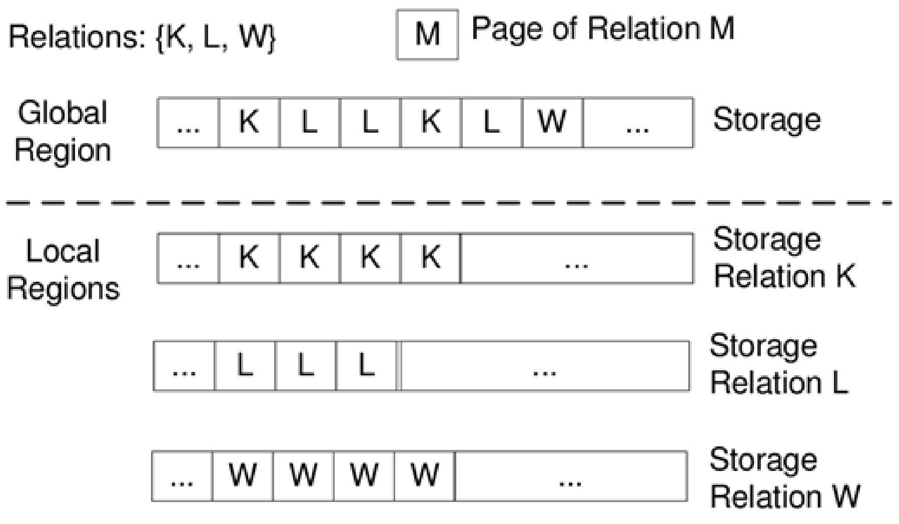


Figure 6.2: Global vs. Local Append Region(s). One global append region comprises appended pages of different relations. One local append region per relation comprises pages of exactly one relation.

6.4 LbSM Evaluation

The evaluation focuses on the I/O characteristics and behavior of the Flash storage technology. It is based on our own work, published in [35], augmentations are made where appropriate.

We compare in-place storage management and page-/tuple LbSM approaches in conjunction with multi- versioning databases on new storage technologies (Flash storage) and elaborate the influence of one single or multiple append regions. Our findings show that while page-based LbSM approaches are better suitable for new storage technologies, they can be optimised by implementing tuple-based LbSM directly into the MV-DBMS. SIAS is implemented (Section 4.3), as a tuple LbSM within the PostgreSQL MV-DBMS which algorithmically generates local append behavior. SIAS leverages the characteristics of Flash storage and addresses their special properties. SIAS achieves high performance, scales almost linearly with growing parallelism and exhibits a significant write reduction. Our experimens show that:

- Traditional LbSM approaches are up to 73% faster than their in-place update counterparts;
- SIAS tuple-version granularity append is up to 2.99x faster (IOPS and runtime) than in-place update approaches;
- SIAS reduces the write overhead (amount of issued write requests) up to 52 times;
- SIAS using local append regions is up to 5% faster than using one global append region for all relations.

Page LbSM as well as tuple LbSM is evaluated in combination with global region and local region append. For page LbSM two additional variants are evaluated, one using garbage collection, marked with the postfix "-GC", and one that omits the garbage collection (no postfix).

For the tuple LbSM, two additional variants are evaluated. An *optimistic* approach that assumes that SIAS data structures are cached, marked with the postfix "-O" and a *pessimistic* approach which fetches the data structures separately, marked with postfix "-P". A variant with/without garbage collection delivered congruent performance, therefore this distinction is omitted here. The complete (100%) filling degree is assumed as the threshold for the evaluated SIAS approaches. This assures that each page gets completely filled before it is appended to the storage. Hence the results indicate the maximum benefit of the SIAS approach. The results of the prototypical implementation vary as the threshold is chosen differently, for example as a variable filling degree, timeout or as the amount of finished transactions. Our key figures and trends, reported in this Section, are confirmed by the implementation, this is further described in Section 4.3.

For comparison the in-place approach, default in PostgreSQL, is also reported and instrumented with and without garbage collection.

Garbage collection (GC) is either activated or turned off for the measurement of the I/O behavior. This experiment is conducted in order to examine the influence of GC on append storage management.

- **SI-PG-(GC)**: Page LbSM that appends pages to one global append region on the storage device.
- **SI-PL-(GC)**: Page LbSM that appends pages to multiple local append regions. SI-PL partitions the global append storage into multiple *local* append regions, dedicating each region to a single relation of the database.

- **SIAS-L-(O/P)**: Multiple/local append regions, where each region is dedicated to exactly one relation of the database. All pages in a local append region belong to the same relation and all tuples within a page belong to the same relation. Tuple versions are appended to a page of their relation. When it is filled completely or has reached a certain threshold it is appended to the relation's local append region.
- **SIAS-G-(O/P)**: One append region for all pages. Tuples within a single page belong exactly to one relation. Tuple versions get appended to a single page (analogously to SIAS-L) which is then appended to a global append region. The global append region maintains pages of all the relations of the MV-DBMS.
- **SI-(GC)**: Default PostgreSQL in place storage management, with (-"GC") and without garbage collection.

Figure 6.3 depicts the evaluated storage manager variants.

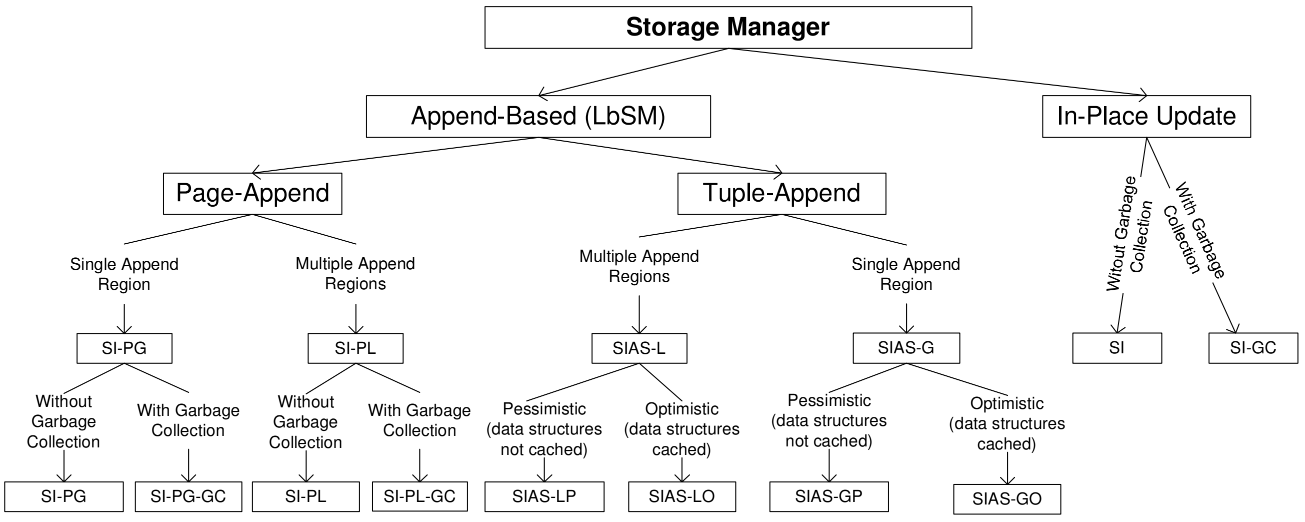


Figure 6.3: Evaluated Storage Manager Variants

6.4.1 Simulation.

The following evaluation of the different LbSM alternatives (depicted in Figure 6.3) is based upon a trace driven simulation. Simulation is chosen for the following reasons:

- (a) Ability to focus on the main characteristics of the multi versioning algorithms (i.e. exclude influences of the transaction-, storage- and buffer-manager as well as PostgreSQL's specific overhead). By setting probing points accordingly within the transaction- and buffer-manager, we eliminate their influence and are able to simulate the raw I/O of heap-data (non-index data) tuples. RAW access to the storage device is used to shield the evaluation from the influence of a filesystem.
- (b) Ability to compare a number of LbSM and SIAS alternatives with the same workload (trace).

- (c) Investigating the maximum performance benefit of each approach.

The simulation (Fig. 6.4) comprises the following steps:

- (i) Recording of the raw-trace;
- (ii-a) Simulation of SIAS and SI resulting in I/O traces;
- (ii-b) Remapping of the traces, creating the appropriate I/O workload requests as I/O traces;
- (iii) I/O trace execution on a physical SSD (Intel X25-E SLC) using FIO;
- (iv) Validation using our SIAS-Vectors & SIAS-Chains prototypes installed on the same hardware [33].

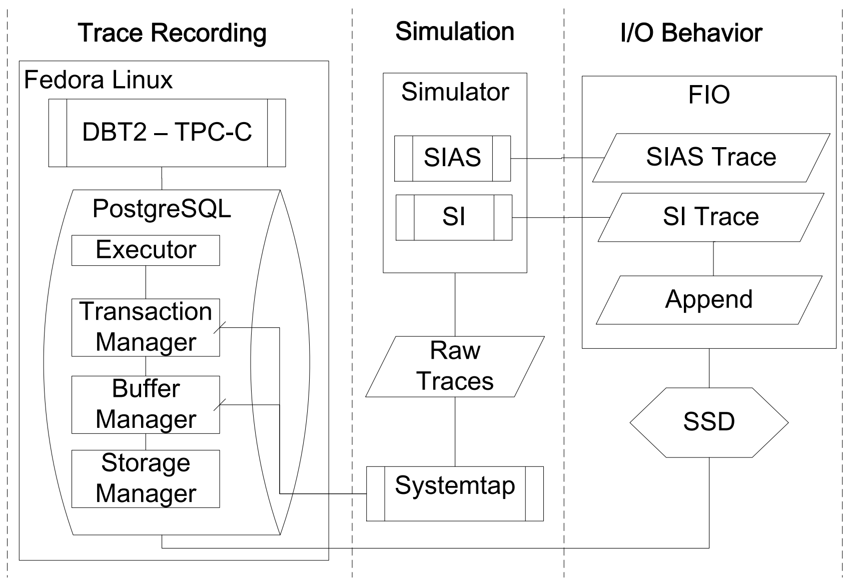


Figure 6.4: Simulation Process. First the trace is recorded using a standard PostgreSQL implementation with probing points set by systemtap. Second the simulator creates I/O traces as they would result in the implementation. Third the I/O traces are fed to the FIO benchmarking tool.

SI and SIAS are simulated based on the raw trace including visibility checks and the resulting storage accesses. During the simulation in step *(ii-a)* and *(ii-b)* the DB state is recreated according to the respective approach (Figure 6.3). The simulated databases always contain exactly the same tuples as the original DB, with the only difference of the permutation of the tuple versions' location. Tuple versions reside on different pages within the simulated DB, according to the corresponding approach. The SIAS approach algorithmically generates a local append and corresponds to the SIAS-L variant. The simulation of SIAS-L gives the maximum benefit of the SIAS approach as described in Section 4.3.

In order to simulate SIAS-G, an additional mapping is performed - analogously to the page LbSM approaches. This is an implementation choice, since SIAS is already based on local regions which allow us to utilize page number offsets for each relation. Although this is only necessary for the creation of the block-level traces during the simulation, this obviously needs to maintain a (global) page mapping similar to the page LbSM approaches. Nonetheless the overhead of this page mapping is less exhaustive since the amount of appended pages is smaller due to the tuple append granularity.

The output of the simulator are block-level traces in the format used in FIO to generate I/O workload patterns. These reflect the I/O access pattern that a MV-DMBS, which uses the corresponding storage manager, would execute against the storage. Finally, the block-level traces are executed on a real SSD using FIO, which allows us to precisely investigate the influence of I/O parallelism and raw access (without a file system). The FIO I/O benchmark guarantees controlled trace execution, repeatable results, configurable I/O parallelism and reliable metrics. FIO was configured using the *libaio* library accessing an Intel X25-E 64GB SSD via direct I/O as a raw device. The raw device had no file-system layer in between. To avoid SSD state dependencies a series of 8KB random writes is executed after each run of a trace.

The simulation workload is created by the DBT2 TPC-C implementation. We record each operation on a tuple version and trace the visibility decision for that version. The resulting raw-trace is fed to the simulator.

6.4.2 Validation

The simulator was validated against the SIAS-Vectors and SIAS-Chains implementation using TPC-C conform [113] workload. The simulation as well as the SIAS implementation(s) deliver comparable I/O patterns, reported and demonstrated with SIAS-Vectors in our own work [33].

The write patterns generated by our simulation and the SIAS prototypes are congruent. Figure 6.5 depicts the blocktrace of the SIAS-Chains implementation run on USB, benchmarked by a TPC-C testrun with 12 warehouses and 2 hours runtime. Since the amount of warehouses is limited, most read pages are already cached, yet the write access pattern is forced to the USB stick (described in detail in Section 4.3). This shows the read and write pattern of the simulated SIAS-LO approach. Figure 6.6 depicts the blocktrace of the SIAS-Chains implementation, benchmarked on SSD by a TPC-C testrun with 100 warehouses and 300 seconds runtime. Figure 6.7 shows the blocktrace of the SIAS-Vectors write pattern on SSD with active garbage collection and the SI write pattern for comparison.

The read pattern shows the random access caused by index accesses. The write pattern illustrates the append as a thin *swimlane*. This corresponds to the simulated SIAS-LO read access pattern discussed later in this Chapter (Section 6.5, Figure 6.21).

The performance is comparable with both approaches. The write reduction slightly differs in favor of the simulation, since the threshold until a new page is appended is set to a time-limit (piggybacked with checkpoint segments) - the write patterns of the prototype show an interrupted line. Performance of the SIAS prototypes is discussed in Section 4.3.

In terms of I/O behavior and I/O parallelism both (prototype and simulation) achieve:

- (i) compareable performance;
- (ii) congruent write patterns; and
- (iii) comparable net/gross write overhead reduction.

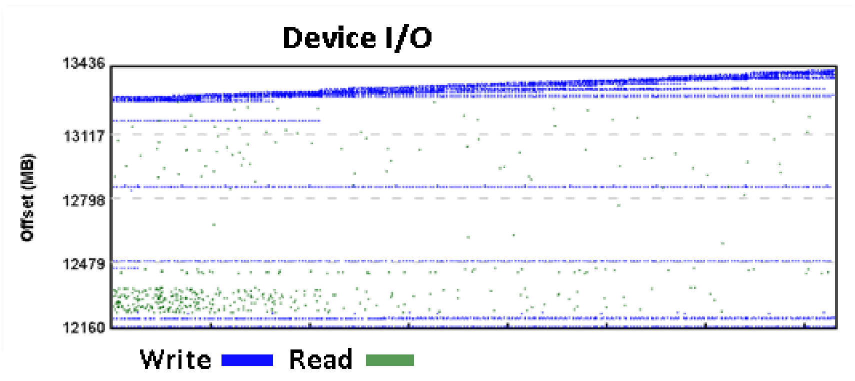


Figure 6.5: Blocktrace: SIAS-Chains on USB - TPC-C - 12 warehouses - 2 hours runtime.

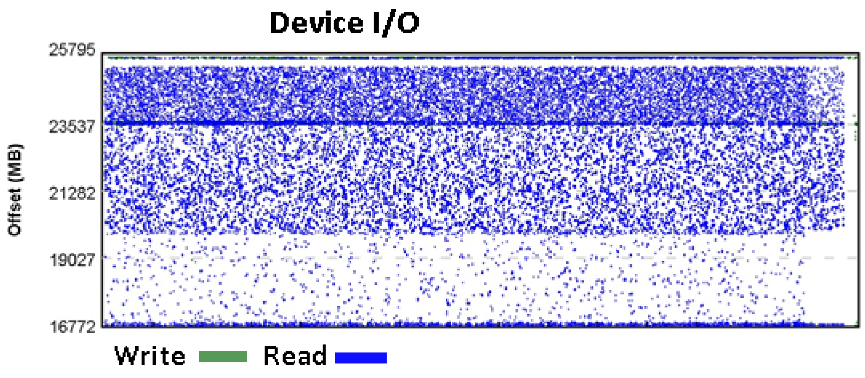


Figure 6.6: Blocktrace: SIAS-Chains on SSD - TPC-C - 100 warehouses - 300sec runtime

Instrumentation.

A default, out of the box PostgreSQL was used to record the raw trace. It was instrumented utilizing TPC-C [113] (DBT2 v0.41 [22]) with the PostgreSQL default page size of 8KB. All physical appends were conducted using this block size. The used Fedora Linux (kernel 2.6.41) included the systemtap extension (translator 1.6; driver 0.152). In order to visualize the I/O patterns the blocktrace of each FIO execution is recorded.

Load Characterization.

The open source DBT2 benchmark v0.41 [22] is used to generate TPC-C conform workload [113]. DBT2 is instrumented to create two traces, both with 10 clients per warehouse and a total of 200 warehouses. *Trace I* with a runtime of 60 minutes and *Trace II* with a runtime of 90 minutes. Based on these two traces we also investigate the impact on space management, chain length etc. The TPC-C testruns for the prototypes are instrumented to show certain effects and characteristic I/O patterns more clearly. The recorded trace in Figure 6.5 is instrumented with only 12 warehouses but a runtime of two hours and serves as an example of the append operation. It allows to adequately investigate the append write I/O pattern, local region append is clearly visible as swimlanes in the simulated approach (also depicted later in Figure 6.20).

The blocktrace of the SIAS-Chains prototype, as depicted in Figure 6.6, is instrumented with 100 warehouses and a short runtime of 300 seconds to firstly emphasize the random read pattern caused by point queries over an index and secondly to depict the amount of appends forming a *swimmlane*.

Raw-Trace.

The raw trace is recorded with the systemtap extension and contains:

- (i) tuple versions and the operations executed on them;
- (ii) the visibility decision for each tuple version;
- (iii) the mapping of tuple versions to pages.

This information is sufficient to simulate the storage manager approaches.

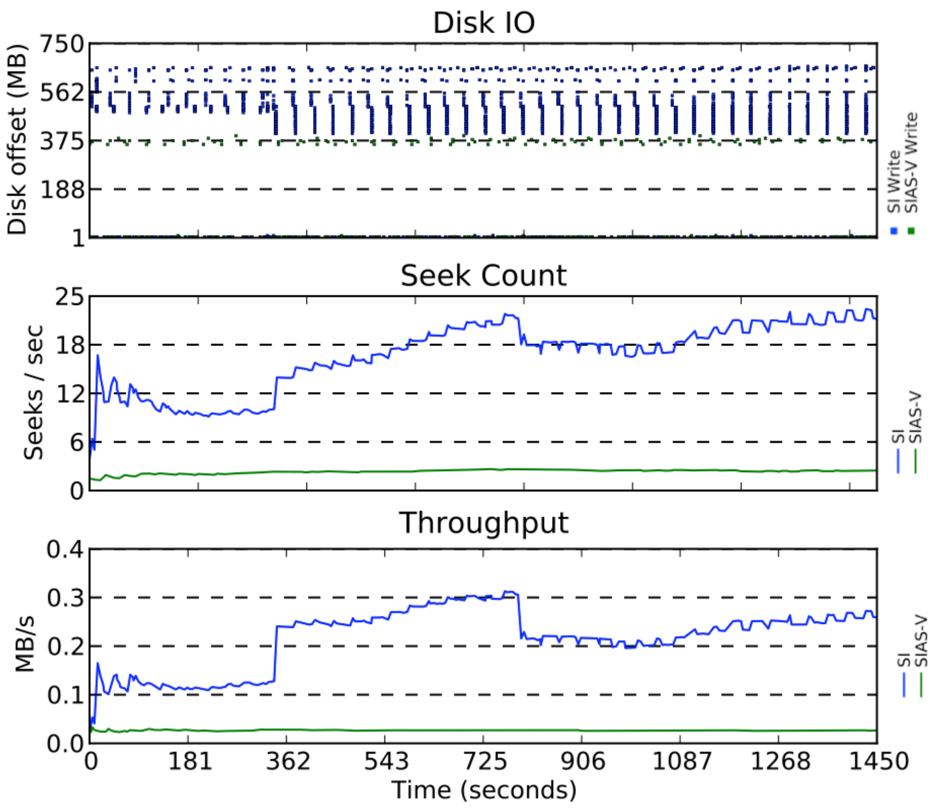


Figure 6.7: Blocktrace on SSD: SI vs. SIAS (SIAS-LO). The SIAS-LO blktrace is recorded using the SIAS-Vectors implementation (Section 4.4).

6.4.3 Results

I/O Performance and Parallelism. Parallelism is examined by instrumenting FIO with different queue depths (QD) ranging from QD=1 (no parallelism) to QD=32 (maximum queue depth of the Intel X25-E 64GB SSD). We measure the IOPS and runtime of each of the storage manager approaches on both traces.

The IOPS of Trace I are depicted in Figure 6.8 and Figure 6.9. The runtime of Trace I is illustrated in Figure 6.10 and 6.11. The IOPS of Trace II are depicted in Figure 6.12 and the runtime in Figure 6.13. The I/O pattern (seeks, throughput, access distribution) for the maximum QD=32 is depicted as a blocktrace in Figure 6.20 and 6.21.

Figure 6.8 illustrates the differences between SIAS and SI-P in both global and local implementation variants. It is clearly visible that SIAS achieves much higher IOPS compared to the SI-P approaches (global and local). Figure 6.9 additionally displays the in-place update approach of SI, which is slower than all other approaches.

- The results show that the in-place storage management approaches show the lowest performance for both traces, independent of the state of the garbage collection (SI and SI-GC). Parallelism is not leveraged.

  The read IOPS of SI-GC (active garbage collection) are the lowest, visible throughout Figures 6.8, 6.9, 6.12 and 6.10.

- Increasing parallelism (QD) emphasizes the performance deficit of the in-place and the LbSM approaches. The in-place approaches can improve up to a QD of two (parallel I/O requests), and stagnate at a QD of four. These approaches do not benefit from higher parallelism.

- Page LbSM and Tuple LbSM scale with a higher QD. Both approaches benefit from higher parallelism.

- The runtime of SI-V and SI-NV is significantly higher (Figure 6.11 and 6.13). SI-GC takes more than twice the time to complete the workload, compared to SIAS-L.

  On maximum parallelism (QD=32) the page-append LbSM approaches SI-PG and SI-PL are up to 73% faster than the in-place update approaches SI and SI-GC.

  Without parallelism (QD=1), the I/O rate of the tuple-append LbSM is approx. 34% higher than in SI-GC and 23% higher than SI. SI is approx. 8% faster than SI-GC. On active garbage collection (GC) SI-PL-GC is 13% faster than the in-place updates approaches and yields up to 25% higher read IOPS than SI if GC is deactivated.

- Tuple LbSMs exhibit higher I/O performance and a lower runtime than page LbSMs. The I/O performance is illustrated by the graphs in Figure 6.8, 6.9 and 6.12. The graphs indicate that the tuple LbSMs increases steeper than the page LbSMs. The runtime is illustrated in Figures 6.10, 6.11 and 6.13, showing an asymptotic trend towards the runtime.

  The runtime of the page LbSM approaches is twice the runtime of tuple LbSM for *Trace I* (Figure 6.10) and *Trace I* (Figure 6.11 and 6.13).

- SI-PG and SI-PL show almost equal performance, yet SI-PL is marginally slower than SI-PG (Figure 6.9 and Figure 6.10).

  SI-PL writes at multiple locations, this requires more write streams (one per each relation) than SI-PG. This is illustrated with a blocktrace in Figure 6.14. The corresponding read blocktrace is depicted in Figure 6.15. The Figure depicts the *seek count* which shows the effect of multiple write streams (higher amount of seeks).

  The append-region for reads/writes of each relation is depicted as a straight, slightly increasing line. With increasing parallelism LbSM approaches using local append regions have the advantage over one global append region.

- Global and local variants of tuple LbSM (SIAS) perform equally well at low levels of parallelism. With increasing parallelism the local approach is approx. 5% faster than the global approach. Figures 6.16 and 6.17 depict the resulting write pattern.
- The performance of the tuple LbSM approach (SIAS) scales linearly with increasing parallelism and benefits from a high queue depth. Runtimes are asymptotically reduced with increasing parallelism. Pessimistic and optimistic SIAS show equal performance, indicating that additional fetches of the datastructures do not significantly impact performance (neither negative nor positive).

Garbage Collection.

- Active garbage collection (GC) negatively impacts the performance of all approaches. This trend is intensified by a higher degree of parallelism.

  On regarding the IOPS in Figures 6.8, 6.9 and 6.12 it can be observed that GC creates significant overhead when using page LbSM. Starting with a queue depth of four, page LbSM approaches loose up to 35% IOPS on activated GC.
- Traditional GC mechanisms are not beneficial for page-append LbSMs, since they *prune (remove)* single tuples and therefore create physical page updates that have to be remapped by the LbSM. Page LbSM with active GC (SI-PG-GC and SI-PL) benefit from higher queue depths but not as much as the variants with deactivated GC.
- Page LbSM approaches with deactivated GC (SI-PG and SI-PL) scale up to the maximum queue depth and experience stagnation on queue depths larger than four (IOPS in Figure 6.8 and 6.12).
- On *Trace I*, SIAS (all variants) is up to three times faster than SI-V, 2.43x faster than page LbSM with activated GC (SI-PL-GC/SI-PG-GC) and approx. 40% faster than page LbSM without GC (SI-PG/SI-PL).
- The performance difference between tuple LbSM (SIAS) using either global or local append regions is marginal and in favor of the local variant, it is therefore not justified to maintain an additional page mapping, as it is necessary for the global variant (SIAS-G).

  The I/O rate correlates with the runtime of the traces and the tendencies observed in this Section are confirmed by both traces (*Trace II* shows comparable results to *Trace I*).
- SI-PL and SI-PG show almost identical runtime behavior as well as SI-PL-GC and SI-PG-GC (Figure 6.11). SIAS is faster than the other approaches (all implementations).

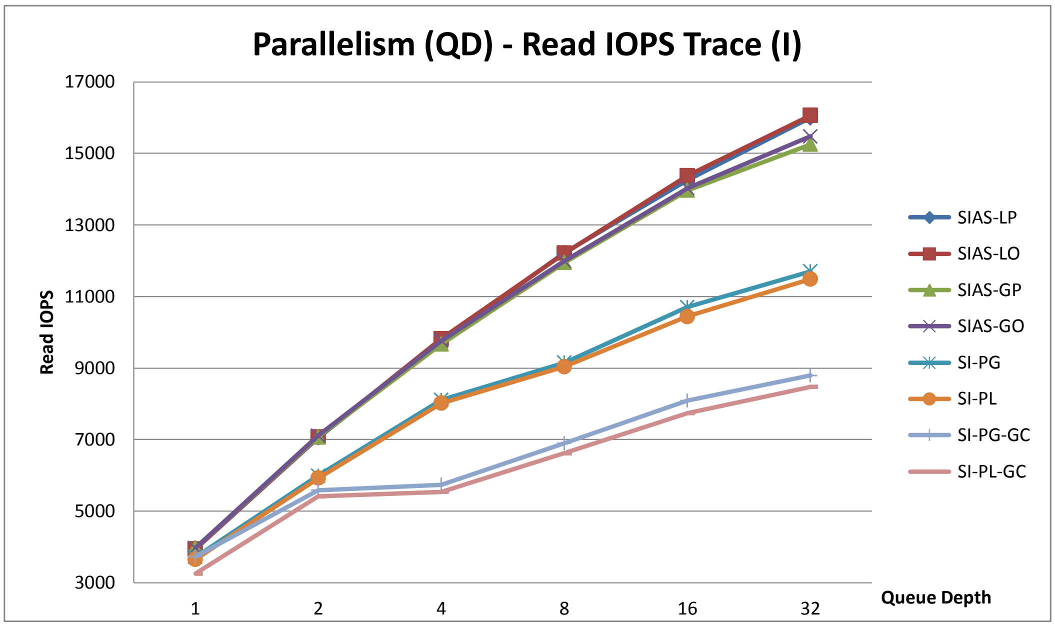


Figure 6.8: I/O Parallelism: Read IOPS vs. Queue Depth for *Trace I*. Tuple vs. page append: differences between SIAS and SI-P in both global and local implementation variants. SIAS achieves much higher IOPS compared to the SI-P approaches (global and local).

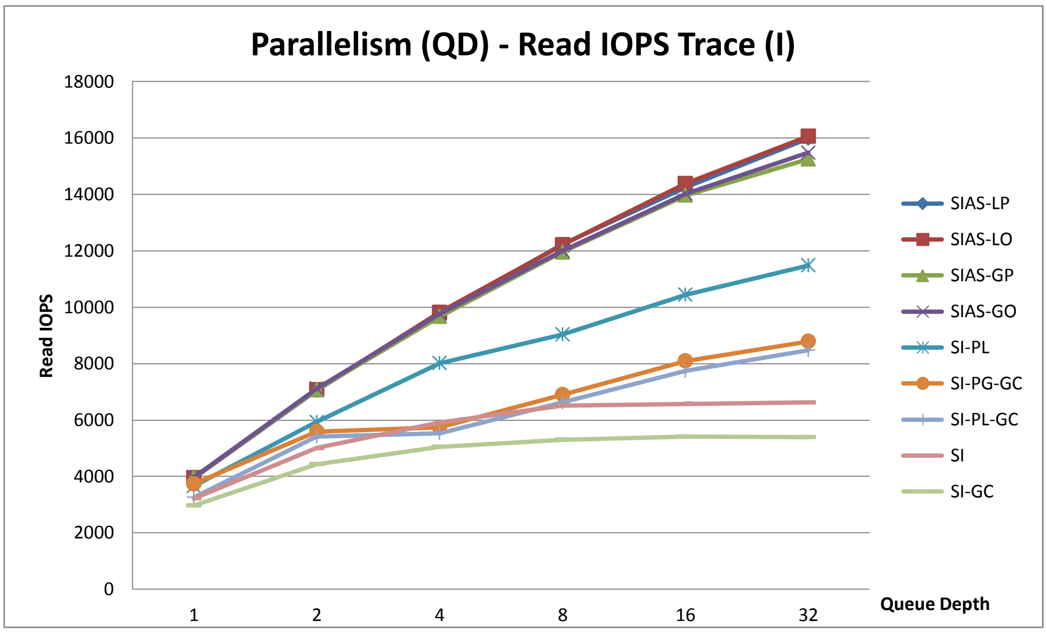


Figure 6.9: I/O Parallelism: Read IOPS vs. Queue Depth for *Trace I*. Tuple vs. page append and the in-place approach. SIAS achieves much higher IOPS compared to the SI-P and SI approaches.

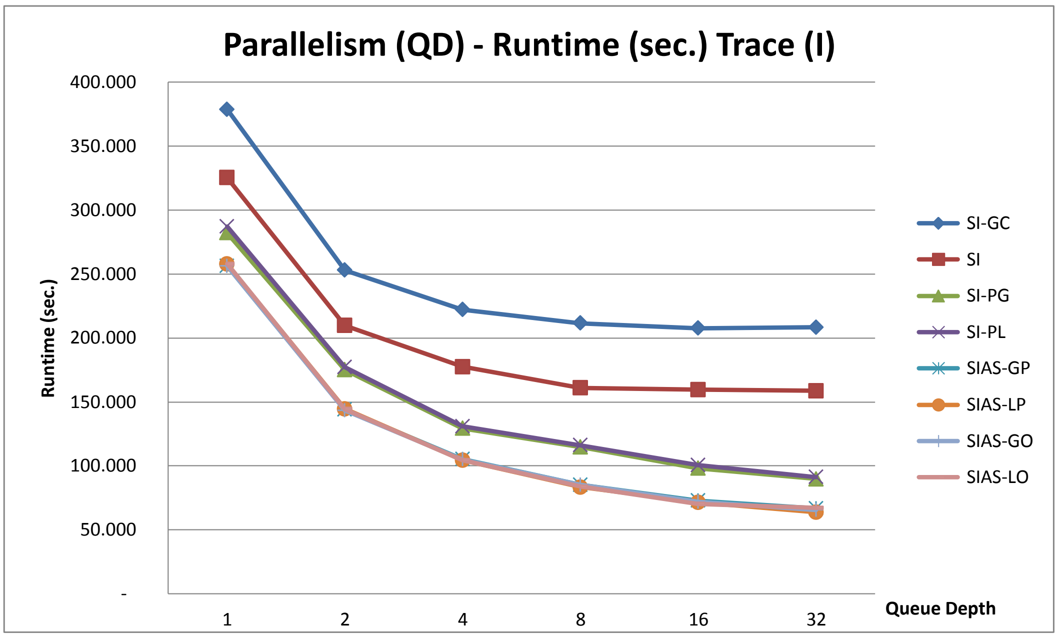


Figure 6.10: I/O Parallelism: Runtime vs. Queue Depth for *Trace I*. Tuple vs. page append. All approaches need less runtime with increasing amount of queue depth. SIAS needs less runtime than the other approaches.

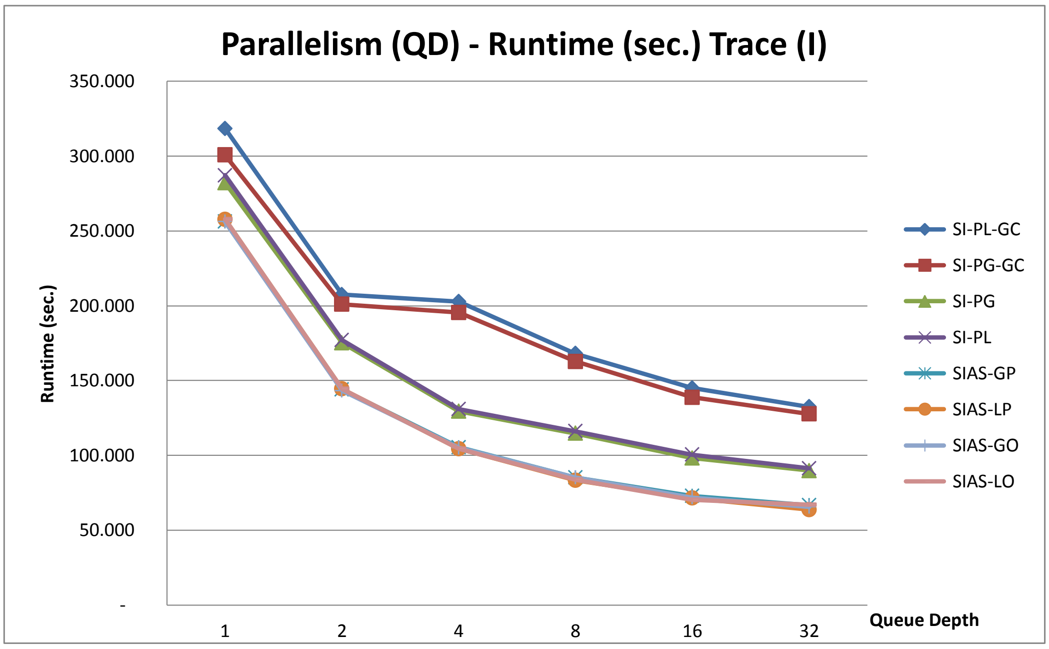


Figure 6.11: I/O Parallelism: Runtime vs. Queue Depth for *Trace I*. Tuple vs. page append and the in-place approach. All approaches need less runtime with increasing amount of queue depth. SIAS needs less runtime than the other approaches. Traditional in-place approaches need a higher runtime.

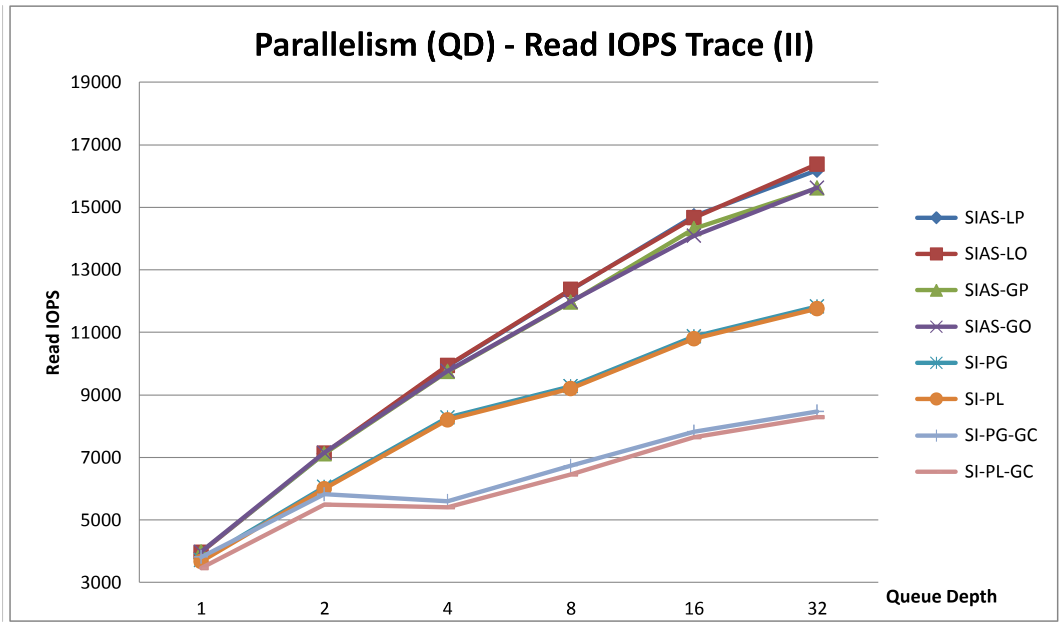


Figure 6.12: I/O Parallelism: Read IOPS vs. Queue Depth for *Trace II*. Tuple vs. page append. SIAS achieves much higher IOPS compared to the other approaches. Global page append with active GC achieves the lowest IOPS.

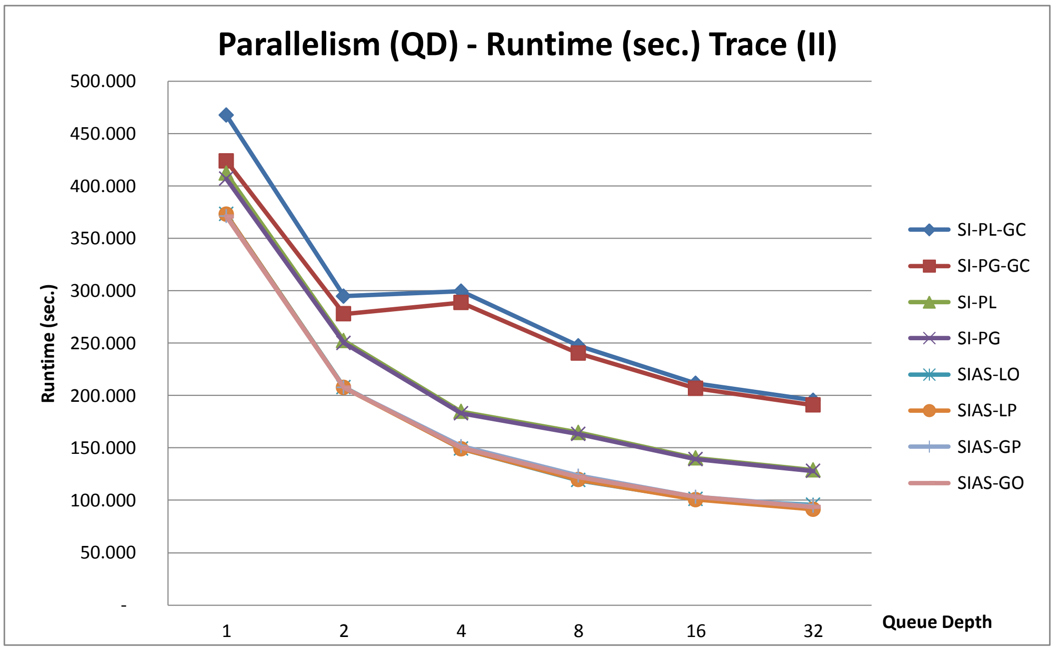


Figure 6.13: I/O Parallelism: Runtime vs. Queue Depth for *Trace II*. Tuple vs. page append. SIAS needs less runtime compared to the other approaches.

Read/Write Overhead.

- In-Place update approaches as well as the page LbSM have the same amount of writes, since the page mapping is performed outside the MV-DBMS. With deactivated GC these approaches (SI-PG, SI-PL and SI) write 966MB in *Trace I* and 1396MB in *Trace II*. With activated GC the write overhead increases. The approaches (SI-PG-GC, SI-PL-GC and SI-GC) write 1304.6MB in *Trace I* and 1975.3 in *Trace II*, respectively.

  This is caused by remappings and relocations of pages issued by the GC.

- Write amplification is significantly reduced by the tuple LbSM approach (SIAS) as compared to the in-place and page LbSM approaches. Figure 6.18 illustrates the write pattern resulting from in-place storage management and Figure 6.18 that of SIAS with local append regions. The in-place approach obviously writes random updates to single pages, leading to a scattered write pattern. In SIAS each relation has one append position to which new pages are appended.

- A key feature of the SIAS tuple LbSM is the significant write reduction. SIAS page LbSM writes (in all variants) 25MB in *Trace I* and 39.9MB in *Trace II*. The amount of writes is equal in all SIAS variants - only new tuple versions (inserted and updated) have to be written. There is no distinction in the amount of issued write requests when appending to local regions or a single global region.

  Compared to *Trace I* SIAS writes 37x less with deactivated GC and 52x less with activated GC. On *Trace II* SIAS writes 34x less with deactivated GC and 49x less with activated GC.

  The write overhead is reduced to a fragment of the usual amount, which is a direct consequence of the out-of-place invalidation, logical tuple appends and full filling of pages. The amount is also depending on the chosen threshold. In this simulation the threshold is defined by the capacity of each page, hence pages are filled to their full capacity. This effect can be observed in the implementation of the SIAS approach, where we also discuss using two differently timed thresholds (Section 4.5.5).

  It is beneficial to chose the threshold that way since a page, once appended is not allowed to be overwritten. Keeping free-space in appended (and immutable) pages is clearly a waste of storage space, since it is rendered unusable.

- Traditional GC is unnecessary for tuple LbSM. In traditional GC of MV-DBMS space is freed by the deletion of dead tuple versions. The deletion entails an update to the already appended page where the page has to be relocated. In addition the free space can't be utilized unless the page is updated (and relocated) again.

  GC for tuple LbSM is discussed in Section 4.6.

- SIAS avoids metadata updates. The metadata update to invalidate a tuple version in SI leads to an update of the page in which this version resides, although the data-load of that version is unchanged. SIAS only needs to append the new version.

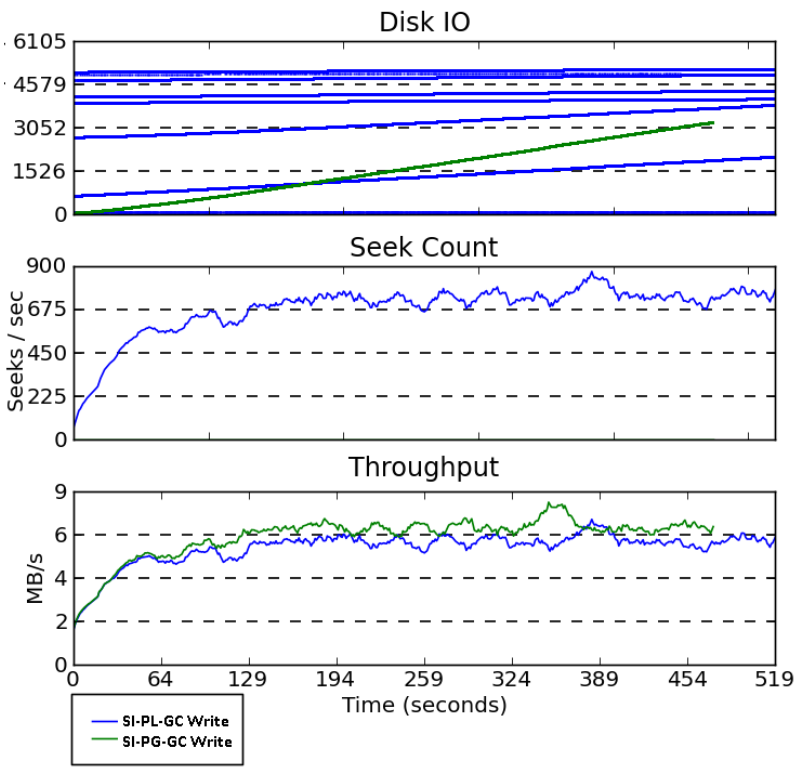


Figure 6.14: Write blocktrace on physical device: SI-PL vs. SI-PG. SI-PG forms one single sequential access, hence not seeks.

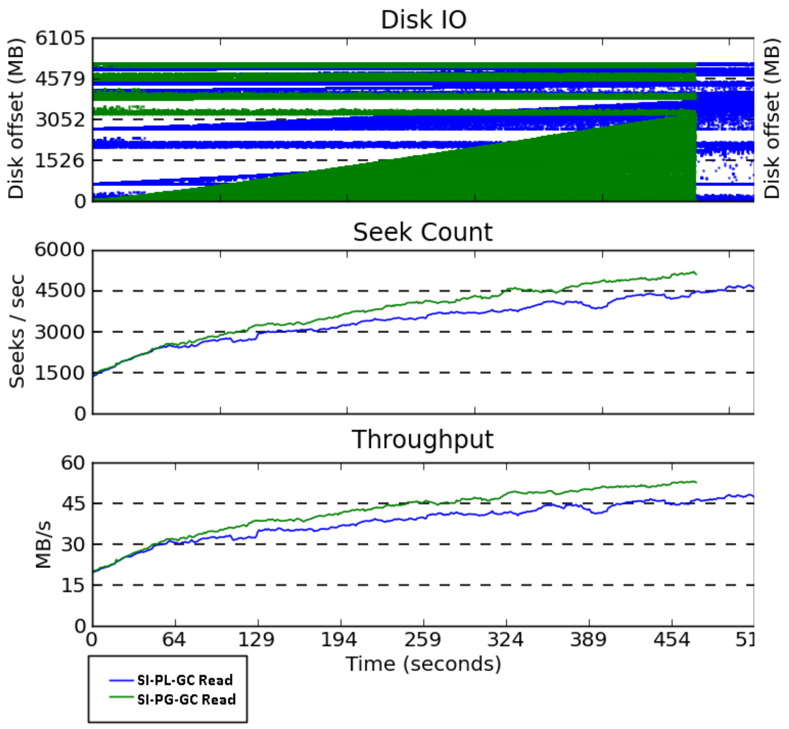


Figure 6.15: Read blocktrace on physical device: SI-PL vs. SI-PG

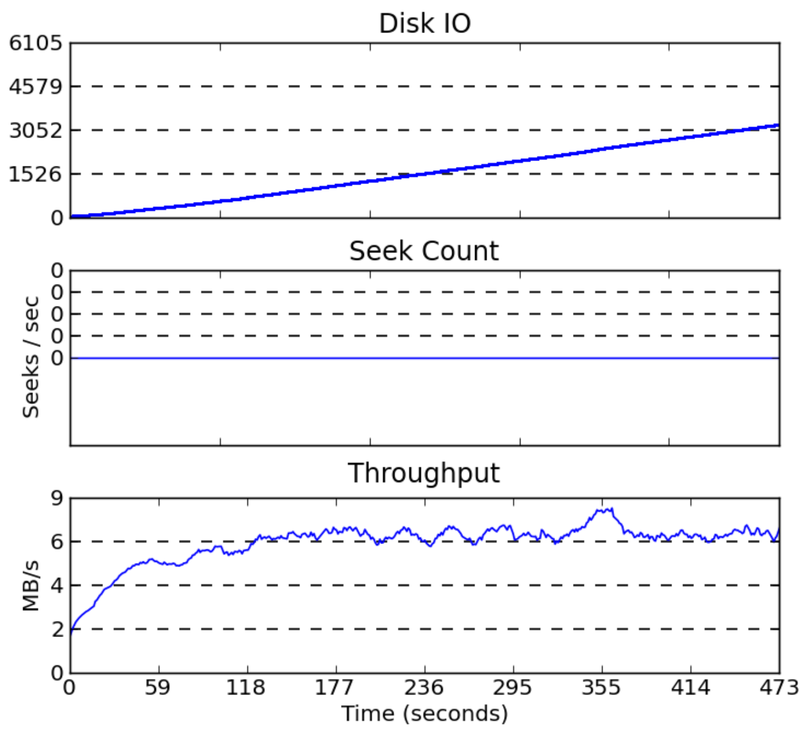


Figure 6.16: Page LbSM: one global append region for all relations. No seeks, since all access is on consecutive block addresses.

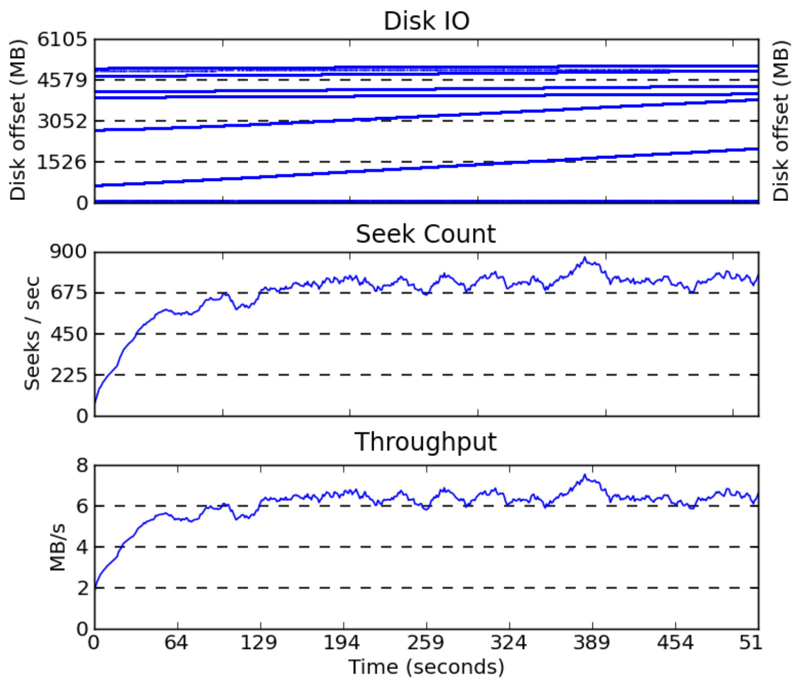


Figure 6.17: Page LbSM: Local append regions. Each append region forms a swimlane.

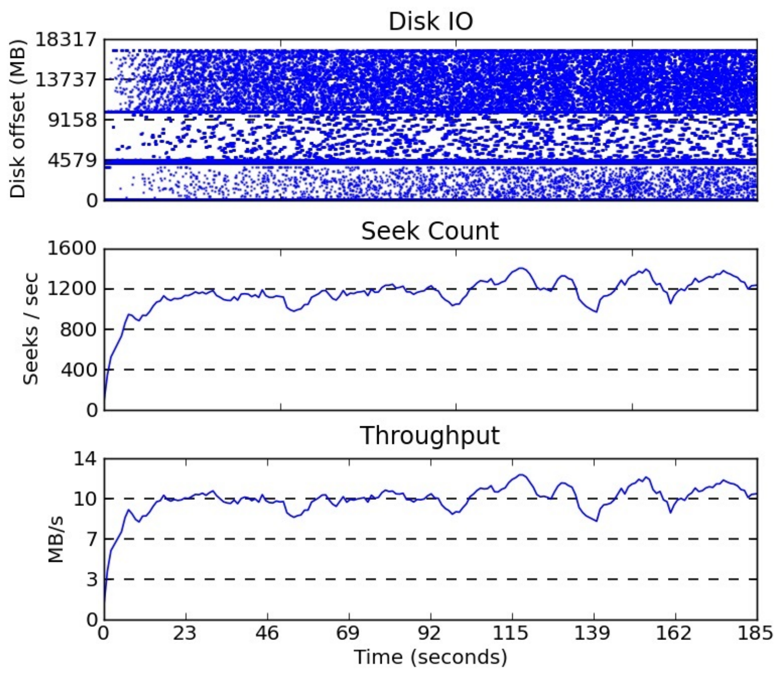


Figure 6.18: Write Pattern: In-place storage management . Write access is scattered on the whole disk. Pages are updated in-place. Large amount of seeks.

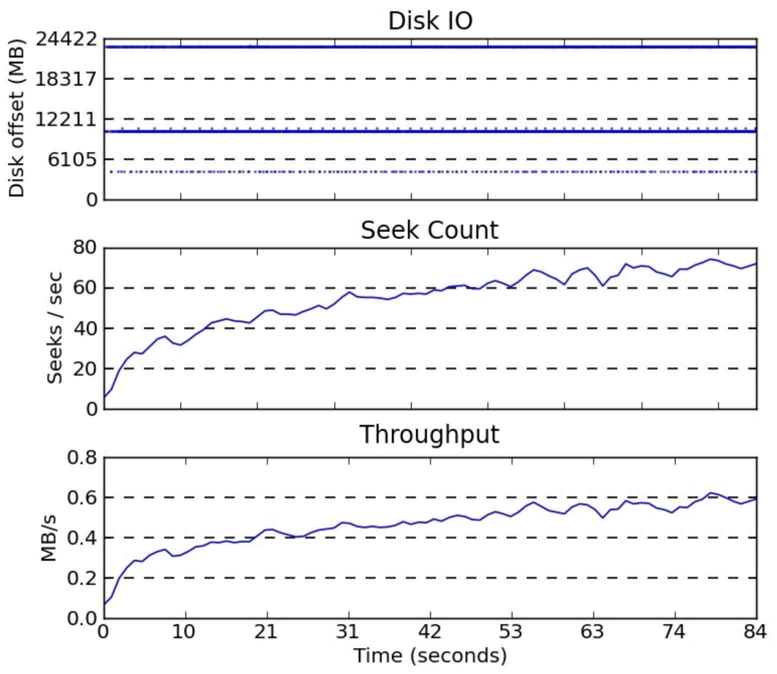


Figure 6.19: Write Pattern: SIAS tuple LbSM with local append regions. Pages are appended forming swimlanes. Low amount of seeks.

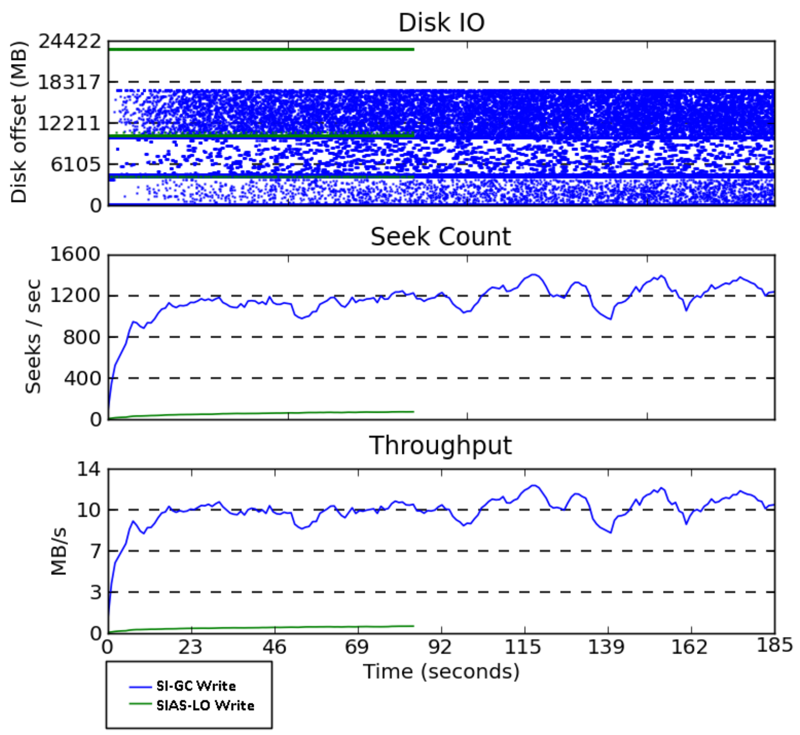


Figure 6.20: I/O Blocktrace on Intel X25E SLC SSD: SI-GC vs. SIAS-LO. SI-GC requires to write more gross data than SIAS-LO. SI-GC issues a large amount of in-place updates. SIAS-LO appends data to each local append region. SIAS completes the trace much faster.

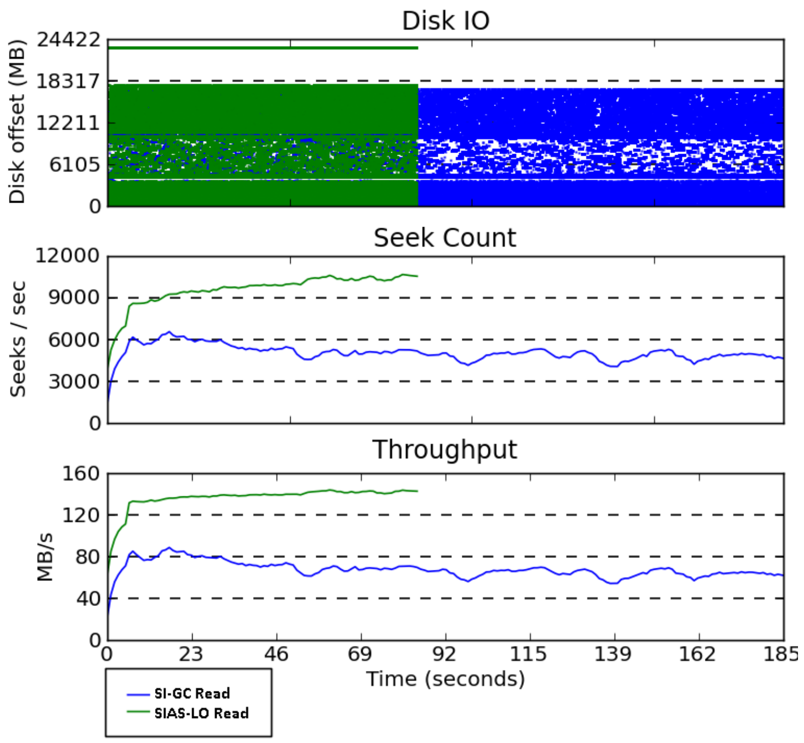


Figure 6.21: I/O Blocktrace on Intel X25E SLC SSD: SI-GC vs. SIAS-LO. SI-GC requires to read more gross data than SIAS-LO. SIAS completes the trace much faster than SI.

6.5 Read Optimizations

This Section discusses an approach to ordered log storage as a read optimization for page/tuple LbSM on Flash, based on our own work published in [38].

Multi-versioning complicates search, both for point and range queries, and modification handling. Search performance is impaired by multi-versioning, since I/O intensive version visibility checks are required for every result set item that matches a search criterion. Modification performance is impaired since OLTP transactions produce a constant stream of new versions (updates and inserts) that result in heavy insert load. Version maintenance overhead such as invalidation of updated versions and removal of invisible versions (garbage collection) causes additional load.

Append storage is a write optimization for Flash storage. The basic principles of ordered log storage are:

- *i)* Enlargement of the portion in the cache that is reserved for each physical append operation and postpone the append until the portion is densely packed with data;
- *ii)* Order the data within each portion according to a sorting criterion.

The first principle increases the cache requirement and influences the cache hitrate. Tuples within a portion have recently been inserted and therefore contain *fresh* data. Since most buffer managers of modern MV-DBMS follow the no-force policy and instrument the recovery manager accordingly (e.g. by using write ahead logging), the durability of the data is not affected.

Optimization: Sorted Runs

The second principle defines the application of a sorting criterion. Each portion represents a *sorted run* and is comprised of multiple pages. The append operation for sorted runs creates an I/O write pattern that is comprised of small sequential writes that are beneficial for Flash storage. Since in page LbSM the data within a page is unknown, the sorting criteria can be made upon the meta-data of each page. In tuple LbSM information about each tuple version is available and the sorting criterion can be more fine granular, e.g. within a sorted run the tuple versions can be sorted by attribute values. Within a sorted run, comprised of multiple pages, multiple tuple versions can be sorted beyond page boundaries. Hence sorted runs contribute to significant table scan speedups.

On subsequent read accesses a sorted run can be read sequentially and the sorting criteria can be leveraged.

The expected benefits of ordered LbSM are:

- Read efficiency due to the use of parallel reader streams;
- Write efficiency since appending in sorted runs creates small sequential writes;
- Faster garbage collection: read multiple sorted runs, filter dead tuples and create one combined sorted run (merge sort).

Optimization: Index Snippets

Index snippets aim at the reduction of tuple version visibility checks, filtering tuple versions which are guaranteed to be invisible to an accessing transaction. Traditional indexing approaches in MV-DBMS are unaware of the versioning scheme underneath and do not store versioning related information, they check the visibility of a tuple version after it has been fetched. The index snippets augment an existing index with versioning information, without changing the index itself. A property of the LbSM is leveraged: Tuple versions within an appended unit correlate according to their creation times. The index snippet augments existing indexing techniques with versioning information about the set of the tuple versions which are contained within the corresponding appended unit, e.g. a single page or a sorted run. The following expected benefits of an index snippet extension are discussed in this Section.

- Visibility Checks on the index level and pre-filtering of invalid/invisible tuple versions, thus reducing unnecessary fetches of pages;
- Postponing of index reorganizations that are caused by versioning (version management);
- Avoidance of *invalid tuple bits* in the index (index in-place updates);
- Fast identification of dead tuple versions that can be garbage collected.

Both optimizations, the *index snippets* and the *sorted runs*, alone aim to accelerate operations along the respective access paths. The combination of both optimizations facilitates index searches on sorted runs. Multi-version indexing in combination with LbSM gains importance, since Append Storage Management inherently disables clustering, thus rendering many read-optimisations ineffective.

6.5.1 Index-Snippet.

Traditional indexing schemes, as inherited by single-version DBMS (SV-DBMS), are capable of returning an immediate answer to an index search, comprised of a set of tuples, a single tuple or nothing. In MV-DBMS tuple versions returned by an index search have to be checked for visibility to the transaction that executes the lookup.

In classic MV-DBMS approaches using SI the visibility is checked by accessing the tuple version itself - which, if it is not already cached in main memory, has to be fetched using a disk access. If the indexed tuple version is not visible to the transaction, the page was needlessly fetched.

Example: under SI a transaction T_x is allowed to see the latest version of a K_v committed before T_x begin (tuple K in version v); if a newer transaction T_{x+1} committed a newer version K_{v+1} after the start of T_x than the MV-DBMS visibility criteria will allow only K_v to be returned to T_x. Traditional index structures contain both tuple versions. If both tuple versions match the search criteria of T_x, both are returned and subsequently have to be filtered. In other words, there might exist tuple versions that match the search criteria but not the visibility criteria, hence a query may take longer to finish.

Definition: An index snippet is a small page-specific part of the index, stored in-memory for every immutable data page that has been appended by the LbSM.

Its structure is as follows (Figure 6.22) :

- A *bitmap vector* that indicates for each tuple version within the corresponding page whether it has a *committed successor* or not.
- A logical timestamp *TP_max* that stores the *commit time of the newest committed tuple version(s) in that page*.

All tuple versions within the page, that have a non-zero entry within the corresponding bitmap vector, have a committed successor in the database, this means tuple versions that are indicated by the bitmap vector have been successfully updated.

If only one tuple version is indicated *TP_max* stores the timestamp on which the update was committed (not the creating transaction's timestamp).

Therefore, *TP_max* abstracts from different transactions and records the most recent update to a tuple version in the page. Transactions that have a begin timestamp greater than *TP_max* are not allowed to access any of the indicated tuple versions. Those transactions have to access a newer (younger) committed tuple version of the data item, which does not have to be the immediate successor of the tuple version.

Transactions yielding a start timestamp smaller than *TP_max* must check each version within the page individually, independently of the snippet's indication. This is necessary since *TP_max* only stores the commit timestamp of the last update.

Therefore the visibility check routines are able to decide some, but not all, checks using the index snippet only.

Yet this facilitates lazy update procedures for other, larger index structures, postponing the deletion of updated versions. Assume an index that stores the location(s) of multiple tuple versions of a single data item - transactions are capable to decide on the visibility without needing to fetch each tuple version from the physical storage. Updated/outdated versions can be left within the index until a certain maintenance operation (garbage collection) is executed, hence post-poning index update operations.

The index snippet speeds up sequential scans on the whole relation that, in case of SIAS, do not use the scan variant of accessing the SIAS-Chains mapping structure first (Section 5.1.2). In general the index snippet improves access methods and paths that access a tuple version first (in comparison to SIAS-Chains that accesses the data item first by using the VID).

Note: In SIAS-Chains this functionality is improved since VIDs (data item identification) instead of TIDs (tuple version identification) are stored in the index.

The implementation of SIAS-Vectors (Section 4.4) implements two bitmap vectors that entail functionality close to that of the snippet. The difference of the implementaion in SIAS-Vectors is the missing timestamp *TP_max* and the semantics of the set/unset bits in both vectors. The combination of both vectors in SIAS-Vectors allows to filter dead tuples, that follow a much stricter criterion than tuple versions that are invisible. A tuple version may be visible to some transactions and invisible to others, hence it is not dead yet. Only if the tuple version is invisible to all running transactions it becomes a dead tuple version.

The traditional approach (see Figure 6.22) follows an (arbitrary) index structure directly to the indicated tuple version, resulting in an access to the page and subsequently the visibility check of that tuple version. The snippet improves on the traditional approach by filtering invisible tuple versions:

- *first*, a transaction follows the index to the row-id of the tuple version;
- *second*, it uses that row-id to check the bitmap snippet and *TP_max* timestamp, both held in memory; either the transaction answers the query immediately returning *false* or;
- *third*, it fetches the tuple version and executes the visibility check, using the tuple version's attributes.

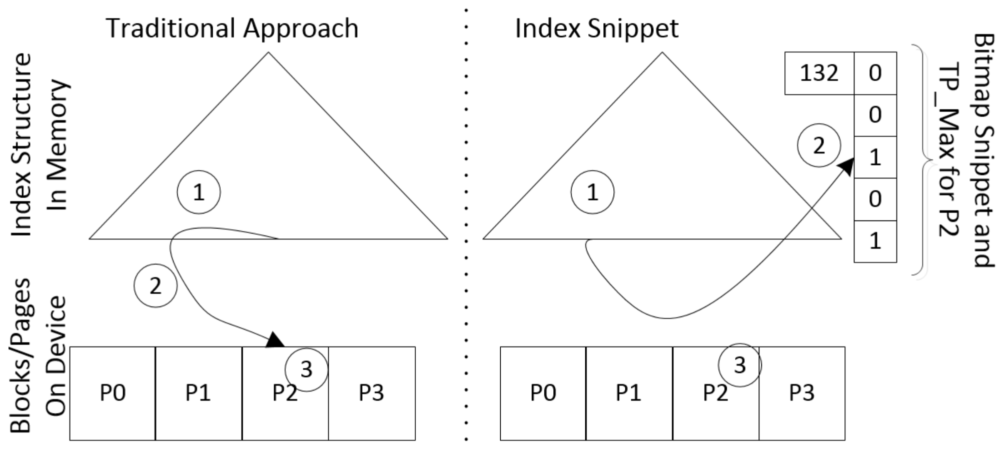


Figure 6.22: Index Snippet. Each index snippet stores invalidation information about tuple versions of a single page in the relation. TP_{max} serves as the boundary for accessing transactions. If a transaction yields a timestamp higher than TP_{max} it is not allowed to access all tuple versions that are indicated by the snippet with a non-zero value .

Filtering property.

The snippet filters invisible tuple versions. It can't be guaranteed that a tuple version is visible to an accessing transaction if it was not filtered. If a tuple version was not filtered it has to be checked for visibility afterwards, even if the bit within the snippet is set. Tuple versions contained in a page can be updated by different transactions which commit at individual points in time, where *TP_max* records the commit time of the most recently updated version. If the bit is set and *TP_max* is greater than the accessing transaction's timestamp, the tuple version may be invisible if *TP_max* was updated by a transaction that updated a different tuple version and committed.

A "false invisible" implies that a tuple version is rejected during the visibility check although it is visible. It is impossible that such a version can exist. Since the *TP_max* timestamp is valid for a committed transaction that updated a tuple in the corresponding page, another transaction that started after that moment in time has to read the newer tuple version. All versions within the page are at least as old as indicated by the *TP_max* timestamp. All indicated tuple versions have a committed successor 'somewhere' in the database system. If the checking transaction yields a begin timestamp greater than *TP_max* it is allowed to discard all tuples in the page that have a non-zero entry in the corresponding snippet.

Snippet Evaluation.

The snippet is evaluated in the SIAS-Vectors approach (Section 4.4) using the more strict criterion of dead tuples. It is not explicitly implemented in the SIAS-Chains approach, since SIAS-Chains is capable of identifying data items rather than versions, hence the issue of multiple versions of the same data item in an index does not occur. Most of the benefits are piggybacked by the SIAS-Chains datastructures, including the dead tuple optimization of SIAS-Vectors. Hence the effects of the snippet are included in the evaluation of SIAS-Chains.

6.5.2 Ordered/Unordered Append Log.

Append/log based storage managers (LbSM) address the problem of slow random writes (small in-place updates) on Flash memories (Section 6.1, also reported in [35]). They eliminate random writes, since each write operation is executed physically as an append.

LbSM obviously destroys the holistic clustering of an entire relation, since an order is not maintained. Asymmetric Flash storage is faster on random reads than on random writes (sustained workload) and the in-place update problem renders a holistic clustering not feasible.

The concept of the *ordered LbSM* is to postpone the flush (physical append) operation of a set of pages and to sort the pages or the contained tuples according to an arbitrary, predefined criterion. The sorting can be defined during table creation time. This creates *sorted runs* analogously to the sorted runs generated by the external merge sort algorithm. It is especially beneficial on Flash, since multiple parallel write and read streams can be used (local append - Section 6.3) .

A prerequisite for building sorted runs is the ability to delay flushing dirty pages out of the database buffer. This operation is facilitated by the no-force policy in modern (MV-)DBMS. The database log and the application of WAL guarantees that no data is lost in the case of a system failure. A combination with group commit can be utilised to maximise the length of sorted runs.

The expected benefits are:

- (i) read efficiency due to leverage of parallel reader streams;
- (ii) write efficiency since larger amounts of data are appended sequentially;
- (iii) fast garbage collection: read multiple sorted runs, filter dead tuples and write a single combined sorted run;
- (iv) smaller mapping table since offsets and inherent compression can be used;
- (v) read optimization, since multiple sorted runs can be read in parallel by the Flash technology;
- (vi) range values reduce can be used to index sorted runs.

Figure 6.23 illustrates the different approaches of ordered LbSM.

Traditional page LbSM is implemented outside the DBMS [109] - *Page Append A* in Figure 6.23. It receives pages that are identified by the layers above using a logical page number, or page ID (such layers assume it to be the physical page number). Subsequently the pages are appended to the physical storage media. During that progress the logical database pages are mapped to physical pages (logical block address - LBA - to physical block address - PBA) using a mapping table to facilitate the append behavior. Pages are not ordered by the LbSM, hence pages are appended in arbitrary order (as received by the DBMS).

The *ordered page LbSM* is depicted in subfigures 6.23.B1 and 6.23.B2. A set of dirty pages is buffered up to a threshold and appended in the logical order as given by the DBMS. The difference between B1 and B2 is the threshold. The size of the sorted run depends on the size of the buffer which is defined by a threshold. In B1 the threshold amounts to four pages and in B2 it amount to two pages, thus influencing the sorting.

Ordered *page LbSM* is unaware of the contents of the page, hence an ordering within single pages has to be managed by the DBMS, which limits the ordering on the granularity (and

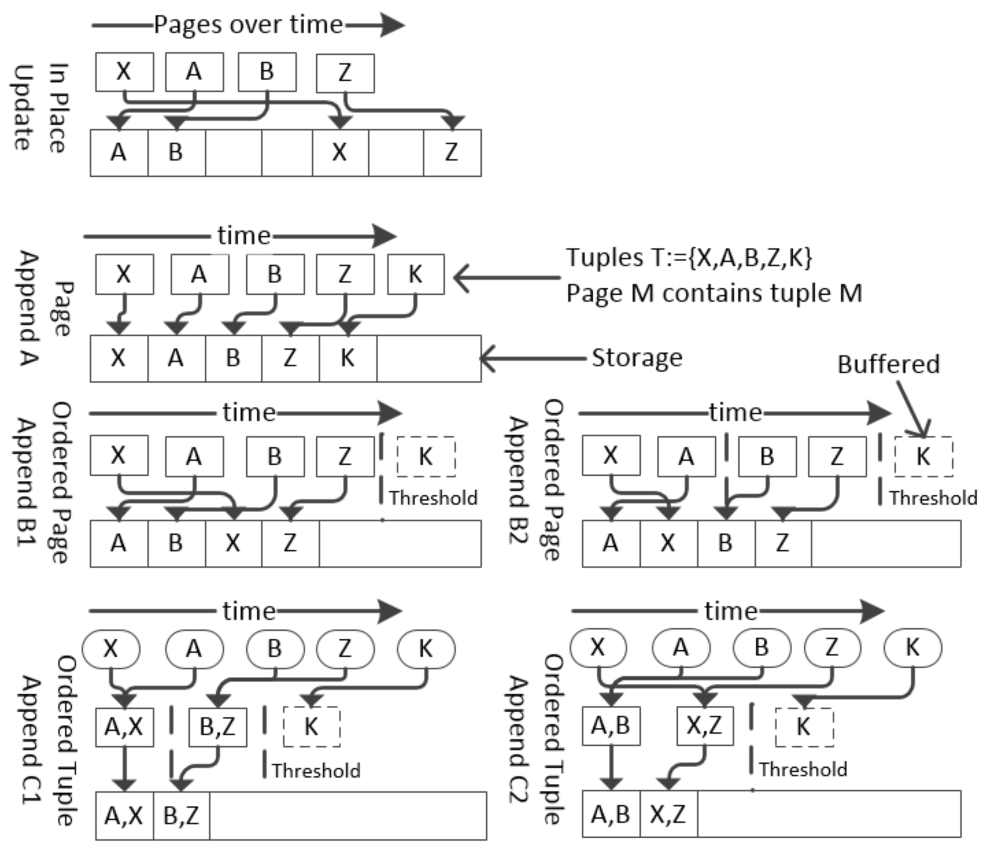


Figure 6.23: Storage management variants. Tuples {X,A,B,Z,K} are stored. In place update managers update each page in place. Page Append A appends each page in the order of the arrival at the storage manager. Ordered page append *B1* and *B2* append a threshold defined group of pages and orders them. Ordered tuple append *C1* and *C2* append tuple versions to a threshold defined amount of buffered pages and orders the tuple versions within that threshold.

properties) of each individual page (no sorting of tuple versions spanning the boundaries of one page).

Ordered *tuple LbSM* append tuples and write in granularities of pages. They enable sorted runs that contain sorted tuples, spanning page boundaries (Figure 6.23.C1 and Figure 6.23.C2). Again a threshold defines the size and the sorting range of a single sorted run.

Figure 6.23.A shows that each page is mapped to a consecutive position on the storage media. Writes are performed in page granularity. For the ordered approaches we use a threshold that defines the least amount of pages to be buffered before a physical write operation is issued. For the ordered page LbSM approaches (Figure 6.23.B1 and Figure 6.23.B2) this defines the amount of pages, whereas for the *ordered tuple LbSM approaches* (Figure 6.23.C1 and Figure 6.23.C2) this defines the amount of individual tuples (circles in Figure 6.23) to be buffered before a write is issued. These changes do not interfere with the write ahead logging mechanism of the MV-DBMS and transactional properties such as recovery are not affected.

LbSM approaches that are based on the granularity of pages (traditional LbSM approach) have no information about the page contents, hence the sorting can only be based on logical page numbers. This means that a sorting of tuple versions cannot span multiple pages. Since the

Table 6.1: Comparison: ordered page LbSM and ordered tuple LbSM

Ordered Page LbSM	**Ordered Tuple LbSM**
append granularity: page	append granularity: tuple version
implemented outside the MV-DBMS	integrated into the MV-DBMS
sorting on page properties	sorting on tuple version properties/attributes
sorting can't span pages	sorting spans entire sorted run
same amount of writes as in-place approaches	reduces write-rate (gross amount)
seperate mapping outside the MV-DBMS	mapping inherently contained in MV-DBMS

traditional approach is located *outside* the DBMS [109], the only sorting criterion is the logical page identification number. A sorted run in this configuration creates a series of consecutive pages, where each logically mapped page receives a physical location according to its individual page identifier. Other pages contained in the sorted run with smaller logical page-ids receive a physical position *before*, and such with logical page-ids higher receive a physical position *after* the page's position.

LbSM approaches that are based on tuple versions (SIAS in [35]) enable sorted runs that span multiple pages. Tuples can be sorted within individual pages and beyond page boundaries. Tuple versions within a single sorted run, comprised of multiple pages, are sorted according to an arbitrary sorting criterion (e.g. value of an attribute). Therefore each sorted run also represents a sorted partition of recently inserted or updated tuples. This co-locates 'fresh' data.

Table 6.1 sums up some important properties and differences of ordered tuple LbSM and ordered page LbSM. In comparison to page append granularity, tuple append significantly lowers the amount of writes, hence reducing the write rate. This is caused by the updates of single tuple versions, since traditional MV-DBMS implement in-place invalidation and follow the paradigm of addressing tuple versions instead of data items. When a tuple version is updated it has to be stamped as invalid which results in the re-write of a whole page. This is avoided by SIAS since the invalidation of the old version is omitted (Section 4.3). The new tuple version is appended at the head of the log and pages are written when they are completely filled - instead of writing up to two pages, one tuple version is appended (as demonstrated in [33, 103]).

Sorted runs - Evaluation.

The approaches are approximated with experiments contained in this Section. For validation the SIAS approach is also included in these experiments. The SIAS approach (Section 4.3) gives practical evaluation results for tuple LbSM that append on local regions using the DBT2 TPC-C benchmarking tool. All experiments are conducted on a physical Intel X25-E SSD by instrumenting the FIO benchmarking tool. We execute different access patterns depending on the specific LbSM algorithm processing the same workload. These experiments abstract from version visibility with the intention of solely identifying the influence of the I/O patterns. Hence all experiments exclude the effects of version visibility checks. Since the ordered append storage is a read optimization, we choose a *full table scan* of a single relation with the size of 5GB as the assumed workload. The resulting physical access patterns are recorded using tools such as *blktrace*, *iowatcher* [51] and *seekwatcher* [102] to visualize the access patterns.

Unordered Append Storage.

The **page append** storage manager *outside* the DBMS receives *flushed* pages from the DBMS. With respect to the algorithmics of a traditional LbSM outside the DBMS [109], a physical order equal to the DBMS' logical page order cannot be recreated. This is due to the required remapping caused by the append operation. Therefore a sequential read access is transformed into multiple random read accesses.

The **tuple append** storage manager, integrated in the MV-DBMS, is capable of identifying a sequential scan. The SIAS approach, as described in Section 4.3, implements such an unordered tuple LbSM. Instead of semi sequential reads, this approach creates a random read access pattern. Therefore this evaluation also executes random reads beneath the sequential access for comparability. Section 6.1 and Section 4.3 show that the amount of written data is efficiently reduced when using tuple append granularity (published in [35, 36, 33]), therefore the tuple append storage manager tests are conducted with an amount of 1% additionally written data.

The page append LbSM as well as the tuple append LbSM both achieve a read throughput of 37MB/s, reading and writing in 8KB pages. As soon as the SIAS tuple append storage manager uses a sequential scan instead of random reads to scan the relation, the performance increases to approx. 123MB/s. The influence of the append at the end of the log has almost no effect on the read throughput. The checking of tuple version visibility is omitted on these experiments in order to focus on the storage management and the resulting I/O. The influence of version visibility checks is conducted in Section 4.3.

Ordered Append Storage.

Approaches performing a sorting of tuples, are capable of identifying a sequential scan since they are implemented inside the DBMS. Approaches implemented outside the DBMS such as [109] have to use separate pattern recognition methods to detect certain access patterns. For the ordered append the relation is split into sorted runs which can be read in parallel. FIO is instrumented with multiple read streams to show this effect. The minimum size of a sorted run is configured as 1MB, therefore the append uses the size of 1MB granularities. This also equals the threshold in the MV-DBMS until pages are flushed. Each page within one 1MB sorted run yields a pagesize of 8KB. Section 6.1 and Section 4.3 (also published in [35, 36, 33]) show that page append outside the DBMS has to write more gross data than the tuple append approach within the DBMS. That effect is evoked in this experiments by reducing the amount of written data for the tuple append LbSM approach (10% relation size for page append and 1% relation size for tuple append).

Configurations.

With the experiments we analyze the effect of the ordered append storage optimization. The experiments are conducted on the following configurations on a single Intel X25-E SLC SSD. The base data was defined by a 5GB relation, yielding a page/blocksize of 8KB. FIO benchmark is instrumented accordingly to execute appends and read accesses at the same time. The workload is defined as a single scan of the relation while appending 10% of the relation size for page append and 1% of the relation size for tuple append.

When the MV-DBMS issues a full table scan, the MV-DBMS is using a forward scan of the relation, fetching the logical blocks with increasing logical blockaddresses (LBA). These I/O requests are subsequently handled by the page append LbSM where each corresponding physical blockaddress (PBA) ist fetched from the storage device. In order to expose the effect of that remapping, the configurations use random reads where appropriate.

- **Configuration 1** represents the traditional page LbSM, implemented outside the DBMS. Random reads are executed and 10% of the relation size is appended (500MB), both in 8KB granularity, using a queue depth of one. Random reads were used to show the effect of the remapping employed by the page append manager.
- **Configuration 2** represents the tuple append LbSM. For comparability to configuration 1 we assumed a simple storage manager which has no knowledge about the executed operation (scan), hence also performing random reads in 8KB (representing the internal mapping in the DB). Simultaneously appends 1% of the relation size (50MB) is appended, since the amount of written data is significantly reduced by such an approach [36, 35].
- **Configuration 2b** represents the tuple append LbSM which has knowledge about the executed operation. It creates a remapping of pages but reads the relation performing sequential block I/O. This configuration yields the same parameters as configuration 2 but using a single sequential read.
- **Configuration 3** represents the ordered page LbSM. FIO is instrumented with different read streams to show the effect of reading sorted runs of pages in parallel. Since each sorted run has a size of 1MB the offset of each run is known which facilitates the parallel request of multiple 1MB chunks. The size of each appended data portion was increased, since pages have to be kept in cache for a longer time. The append builds sorted runs where pages are sorted by their logical block number.
- **Configuration 4** represents the ordered tuple append. FIO is instrumented to read sorted runs of tuples and pages. The append is conducted in the granularity of 1MB units. Each 1MB unit is comprised of pages that have a page size of 8KB.
- **Configuration 5** is equal to configuration 4 with the difference of a larger append unit of 2MB.
- **Configurations 6 and 7** are equal to configurations 1 and 2 with the difference of executing random reads with maximum I/O parallelism (queue depth of 32). This simulates parallel index accesses. This implies that each storage manager has sufficient knowledge about the given workload in order to request the necessary amount of pages.

Configuration 1 (Figure 6.24a) shows a high seek rate, since each read issues a random read while appending new data. Such a pattern is a good match for the SSD, however on a magnetic disk this access pattern would lead to a dramatic performance decrease. The throughput amounts to 37MB/sec on pure read load and increases to 45MB/sec when writing in parallel.

Configuration 2 (Figure 6.24b) is slightly faster than configuration 1 and finishes approx. 8 seconds earlier. The reduction of the appended data portion leads to this effect.

Configuration 2b (Figure 6.26a) shows the bulk read of the whole relation span. Seek rate is high while writing at the same time. This benchmark is rather synthetic, yet an improvement of the bulk read can be observed (approximately 2.5 times). Parallelism is not leveraged by this configuration, it shows the HDD optimized access pattern. Compared with configuration 3 this shows the benefits of the tuple append integration into the DBMS.

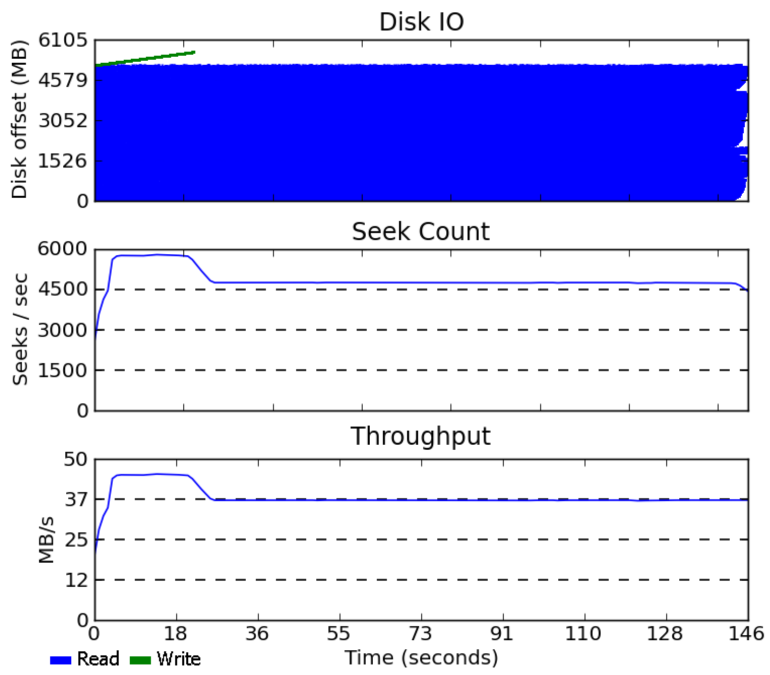
Configuration 3 (Figure 6.25a) requires significantly less time to complete than the first three configurations. This shows the advantage of the ordered page append over the unordered page/-tuple append. No additional seeks are required since there is no mixture with reads. Even a single sequential scan in configuration 2b is not faster than reading multiple sorted runs in parallel, showing the benefit of leveraging the parallelism of the SSD.

A further improvement is given by the ordered tuple append in configuration 4 (Figure 6.25b). The tuple append has to write in larger granularities and inherently creates sorted runs. Increasing the size of the sorted run to 2MB has no significant further improvements on throughput or time to finish, as configuration 5 (Figure 6.26b) shows.

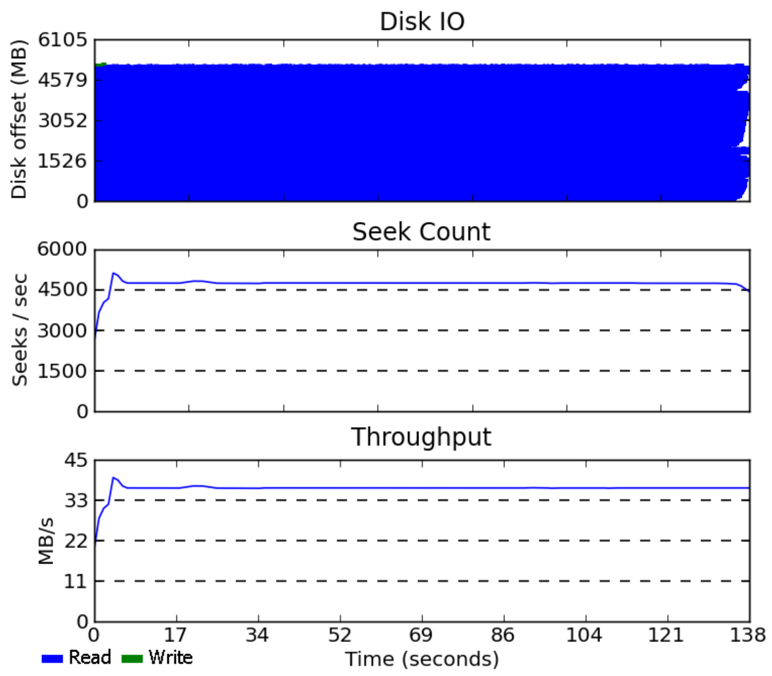
Configurations 6 and 7 (Figures 6.27a and 6.27b) show the expected performance advantage. These however requires a deeper integration of the LbSM into the (MV-)DBMS.

Ordering pages and the content of pages is beneficial for Flash devices such as SSDs. Even with assumptions that favor the traditional approaches, the ordering shows promising results.

With more data skew, which is typical for OLTP systems, where approx. 80% of all accesses only touch 20% of the data the ordered tuple LbSM further improves. We expect the optimizations to be more beneficial for such workload, because of the performed co-location of most recently updated/inserted tuples. This effect has also been observed in our own work in [34]. As an example lets assume the *orders* relation in the benchmark suite DBT2/TPC-C [113, 22]. A sorted run can be configured to contain orders that have sorting attributes such as each *order ID*, thus accelerating subsequent scans.

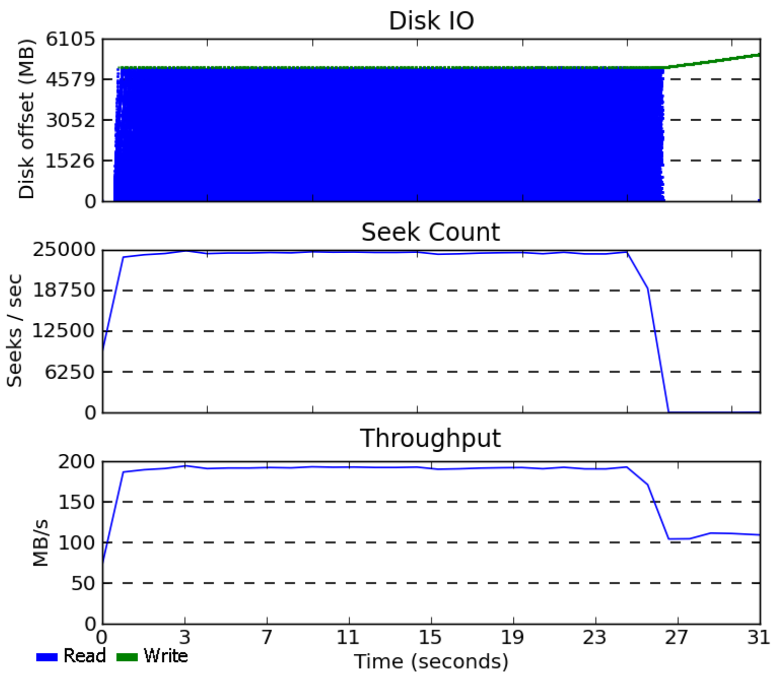


(a) Unordered Page Append - Configuration 1

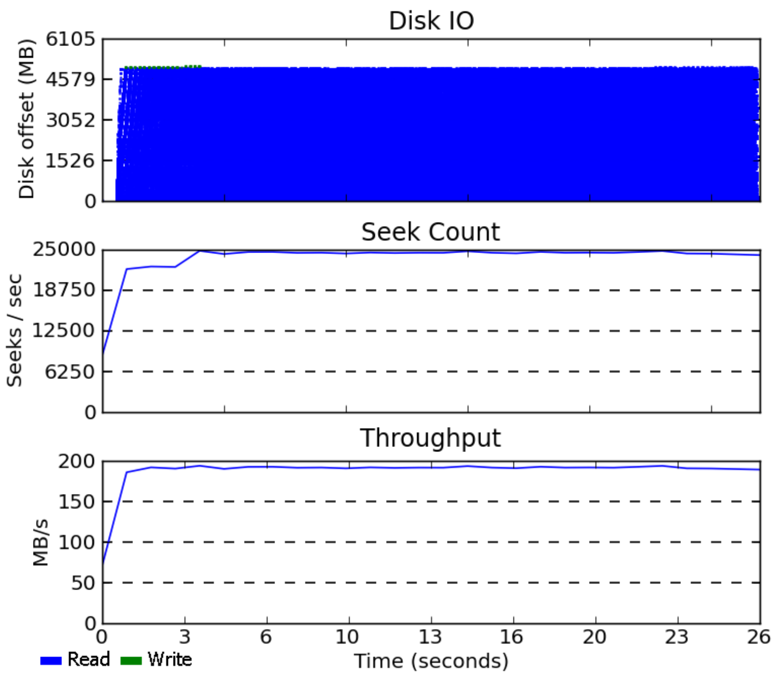


(b) Unordered Tuple Append - Configuation 2

Figure 6.24: Unordered Append - (a) appends in pages, writes a higher amount of gross data and requires more time until the trace is completet. (b) appends in tuple granularity and requires less time until the trace is completet. Both configurations are executed with a queue depth of one.

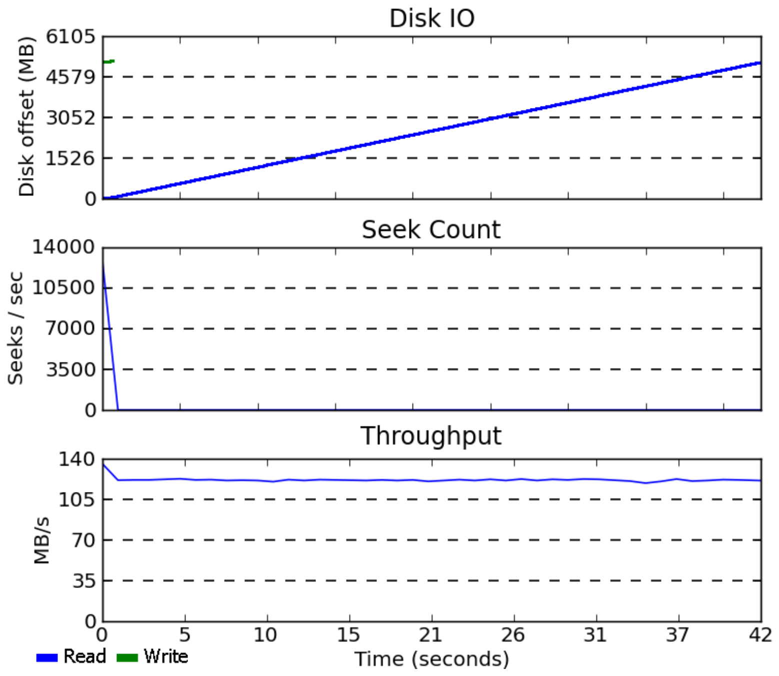


(a) Ordered Page Append - Configuration 3

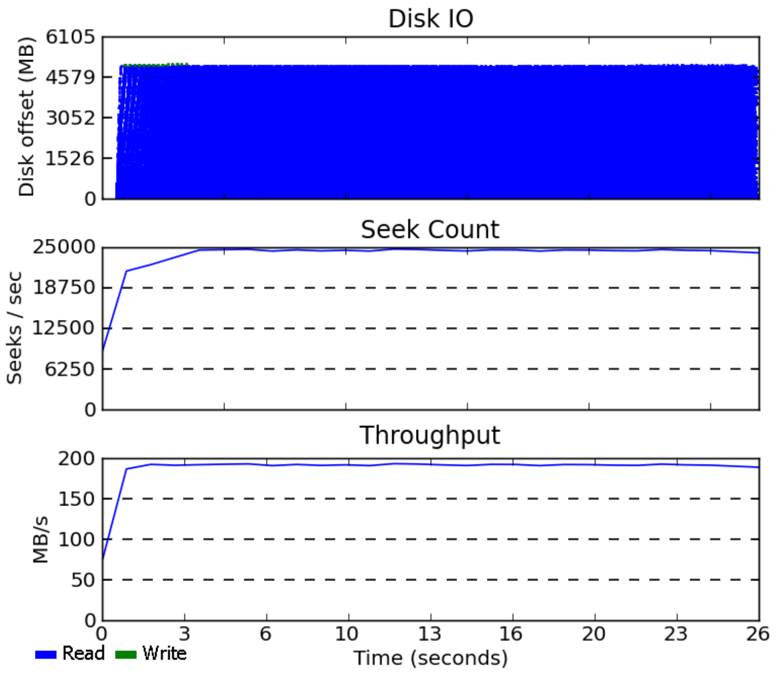


(b) Ordered Tuple Append - Configuration 4

Figure 6.25: Ordered Append - (a) requires more time until the trace is completed since the page append also has to write more data. (b) appends in tuple granularity, requires less completion time and writes less data.

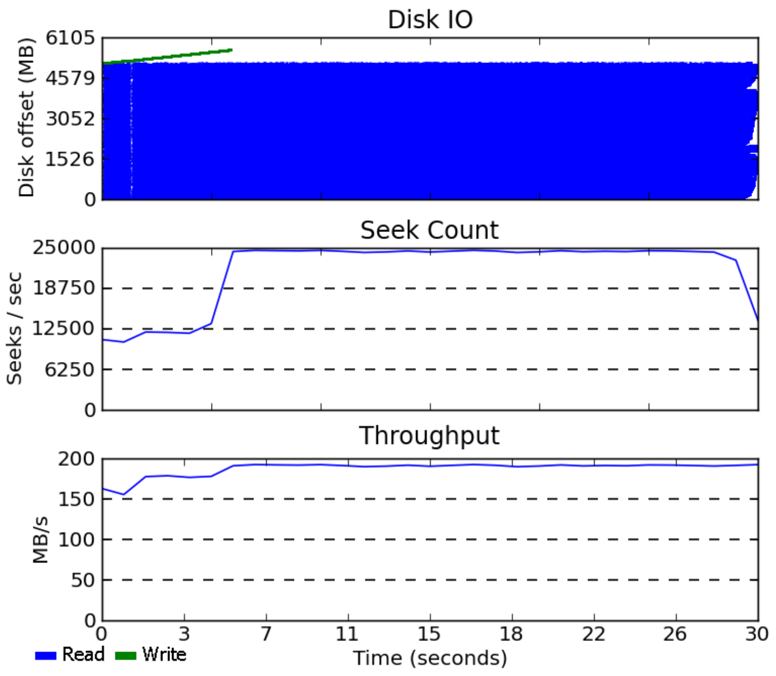


(a) Ordered Page Append - Configuration 2b

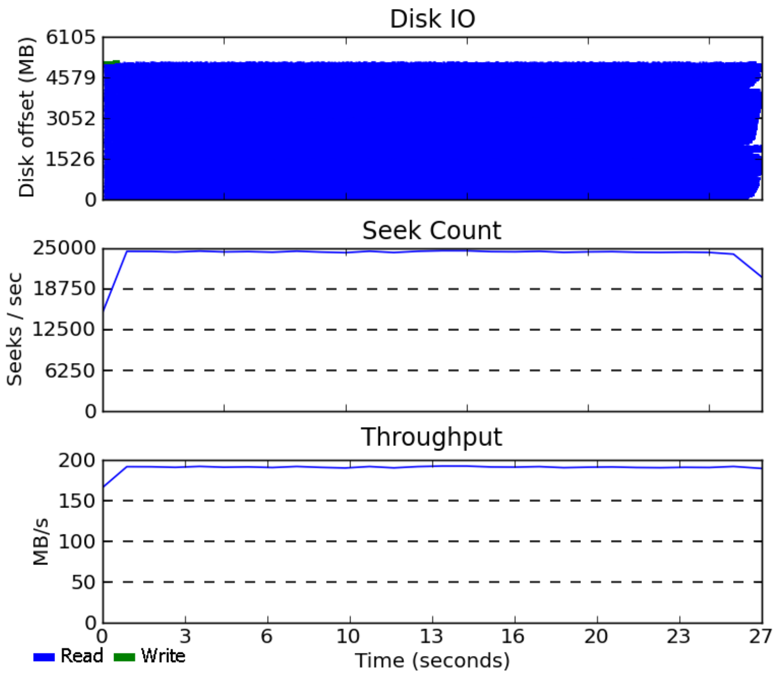


(b) Ordered Tuple Append - Configuration 5

Figure 6.26: Ordered Append. (a) uses a single sequential read operation. Seeks are minimized. This equals a HDD read optimization (single sequential scan). Parallelism of the Flash storage is not used.



(a) Unordered Page Append - Configuration 6



(b) Unordered Tuple Append - Configuration 7

Figure 6.27: Unordered Append - Queue Depth 32. Increasing the queue depth is benefitial for Flash storage. Parallelism can be leveraged, throughput increases up to the Flash SSDs maximum capabilities.

7 Buffer Management

Contents

7.1 Related Buffer Management Approaches . . . 130
7.1.1 CFLRU . . . 130
7.1.2 CFDC . . . 130
7.1.3 FOR . . . 130
7.1.4 DULO . . . 131
7.1.5 BPLRU . . . 131
7.2 Flash Based Adaptive Replacement Cache . . . 132
7.2.1 FBARC - Algorithm . . . 133
7.3 FBARC: Instrumentation and Experiments . . . 138
7.3.1 Raw Trace . . . 141
7.3.2 Workload and Trace Characterization . . . 141
7.4 FBARC: Experimental Evaluation . . . 145
7.4.1 CPU Time . . . 145
7.4.2 I-O Time . . . 146
7.4.3 Overall Time . . . 147
7.4.4 Scan Resistance . . . 148
7.4.5 FBARC Cluster Sizes . . . 149
7.4.6 Behavior on HDD . . . 150
7.4.7 FBARC Summary and Conclusion . . . 150

Buffer Management is a central component of database systems and plays a critical role in transaction management. From an architectural perspective, it addresses the task of minimizing the substantial *access gap* in the latencies of main memory and symmetric disk storage.

Traditional buffer management strategies utilize temporal locality of page accesses to increase the probability of locating the requested pages in the buffer (hit-rate) and to minimize expensive disk accesses. Such strategies utilize two criteria to select victim pages to be evicted from the buffer once new ones are requested: recency and frequency.

The primary criterion of traditional buffer management strategies is hit-rate maximization. Over the last decade asymmetric storage technologies (Flash, Non-Volatile Memories) have emerged with new characteristic properties (Chapter 2) that question the effectiveness and efficiency of present design assumptions of buffer management algorithms. The asymmetric property renders the cost of a page eviction several times higher than the cost of fetching a page. Hence buffer management strategies for modern storage technologies have to address write-awareness and spatial locality besides hitrate.

FBARC is designed to operate independently of SIAS. FBARC operates with in-place storage management, requiring less invasive changes of the DBMS than the SIAS approach. SIAS as a whole compilation of architectural and algorithmic changes, addresses the crucial in-place updates which are still issued by FBARC. In the following related buffer management strategies are discussed and the Flash Based Adaptive Replacement Cache (FBARC) is described.

7.1 Related Buffer Management Approaches

7.1.1 CFLRU

CFLRU [87] is a very light weight extension of the traditional LRU algorithm. It prioritizes dirty pages over clean pages by replacing clean pages at the LRU end of the LRU stack first (the so called priority region). This behavior may be suboptimal with respect to the following reasons. First, the priority region tends to fill up with dirty pages, and thus the constant eviction victim selection runtime of LRU degrades to a linear run time based on the size of the priority region. Second, it affects the hitrate substantially since clean pages are evicted as soon as they enter the priority region, thus effectively lowering the cache size for clean pages. While the size of the priority region can be dynamic it needs a tuning parameter for the cost of a write or, if the priority region is static, it needs a well chosen size based on the workload and thus replacing one tuning parameter with another.

7.1.2 CFDC

CFDC [83] improves upon CFLRU in two ways. It still divides the cache into a work region and a priority region, but it further sub-divides the priority region into a clean and a dirty region. This way it overcomes the search-for-clean-pages problem and allows for a constant search time. In addition, CFDC clusters the dirty page priority region based on the spatial locality of the pages. It still evicts clean pages first, but if there are none then the oldest page from the cluster with the least priority will be evicted and the priority for this cluster is set to zero until it is completely evicted. The priority is computed on every change of the priority region for all clusters and is based on the temporal and spatial locality of the pages within the cluster. This causes CFDC to be very computationally intensive.

7.1.3 FOR

The FOR [73] algorithm approaches the problem from a different perspective. It classifies pages according to two operations: the last completed and the currently executing. Using the resulting ’interoperation distance’ it can recognize a hot page even if it was already evicted, and thus can retain it longer when it is requested again. This approach requires that two data points (last read and last write) are saved for every accessed page over its whole life. Additionally, every operation has to be recorded, i.e. every pin and every dirty flagging has to change the possibly very long operation list. To overcome these issues the authors introduce FOR+ which tries to approximate the inter-operation distance based on the currently cached pages. This approximation works by dividing the LRU stack into a hot and cold region and assigning page weights based on region membership. However, as FOR/FOR+ are built on an LRU basis there may be scan resistance issues. Furthermore, both FOR and FOR+ need carefully chosen cost estimation for reads and writes and FOR+ needs another tuning parameter for the size of its cold page index.

7.1.4 DULO

The DULO [52] cache replacement algorithm tries to combine both the temporal and the spatial locality on top of a LRU stack. It was designed for rotating disks and thus tries to reduce the amount of random accesses in favor of sequential prefetches. It divides the LRU stack into three parts: the working area at the MRU end which consists of single pages, the fixed size sequencing bank and the sequential area at the LRU end which consists of sequential page sequences. The sequencing bank is used to create sequences of pages that can be evicted at the same time and can be read in later with the help of prefetching. To reduce the sequencing cost DULO uses the CLOCK algorithm, which allows it to do the sequencing only when an eviction is needed instead of doing it on every access that causes the LRU stack to change. At eviction time DULO then evicts a complete sequence of pages from the LRU end. This works well on a system that uses a lot of sequential read accesses, but as prefetching does not work for writes it is difficult to achieve write awareness. On flash based storage DULO is not as efficient, as random reads are not much slower than sequential reads.

7.1.5 BPLRU

BPLRU [62] is an embedded device cache level replacement algorithm. It works on a block level and targets only the write buffer on a flash device assuming a log-block FTL [20]. As all its pages are dirty and ready to be written at any time it doesn't have to keep any "clean"-hitrates high. As any hit on a page within a block moves the complete block to the MRU position this can lead to a decreased hitrate as older pages are kept longer in the cache. As a protection against cache flushing it uses a sequential write detection algorithm, which puts sequentially accessed blocks directly to the LRU position. When a block has to be evicted BPLRU attempts to perform a switch merge and pads the block if needed, which reduces the amount of the expensive full merges. Another embedded replacement algorithm that relies on spatial locality and focuses on write caches is WOW [32]. In contrast to BPLRU it was designed for hard disk storage controllers. A cache at that point in the storage hierarchy is is non-volatile (battery backed) and limited by the available resources, but also lacks the information of the current hardware state. WOW unifies CScan – an elevator algorithm for spatial locality, with LRW, an LRU adaption for write caches. The writes are ordered sequentially and grouped in fixed size write groups. This write groups are then put on a ring in a CLOCK like manner and then are evicted in sequential order. A write group can stay in the cache if it is marked as hot when the pointer passes it, but has to be remarked as hot to continue staying in cache.

7.2 Flash Based Adaptive Replacement Cache

This Section is based on our own work [29]. The Flash Based Adaptive Replacement Cache is an extension of the adaptive replacement cache (ARC [77]). FBARC extends ARC by a write list utilizing the spatial locality of evicted pages to produce semi-sequential write patterns.

FBARC is a *special purpose, clustering* buffer management algorithm, which partly uses *page state* information. FBARC relies on ARC as a general purpose algorithm. It features a distinction between recency and frequency and can automatically adapt to workload changes. The ARC [77] replacement policy operates with two lists, one that represents recency and one that represents frequency. FBARC extends ARC by a write list utilizing the spatial locality of evicted pages to produce semi-sequential write patterns. For this purpose FBARC adds an additional list to host dirty pages grouping them into fixed regions (clusters), based on their disk location. FBARC is designed to operate with in-place storage managers. In comparison to other buffer management approaches (CFLRU [87], CFDC [83] CASA [82], FOR/FOR+ [73], [57]), FBARC:

- addresses *write-efficiency* and *endurance* of asymmetric storage;
- offers comparatively high *hitrate*;
- is *computationally-efficient* and uses static grid-based clustering of the page eviction list;
- adapts to *workload changes*;
- is *scan-resistant* - this property is inherited from ARC.

Since traditional storage technologies exhibit symmetric performance (i.e. reads are as fast as writes, Figure 7.1), maximizing the hitrate thus minimizing disk accesses of whatever type (read or write) is a sufficiently general, single design criterion.

The characteristics of asymmetric storage technologies have significant impact on the buffer manager and influence the design goals for a buffer management strategy. In face of the shrinking and operation dependent (read/write is asymmetric) access gap, with the FBARC approach we claim that addressing I/O asymmetry should be considered as first-class criterion besides hitrate. The write-efficiency aspects of I/O asymmetry can be addressed by utilizing *spatial locality* during the page eviction process.

Existing approaches rely on *write region clustering*, which may result in high computational costs.

The design trade-off, therefore is to minimize eviction costs at the expense of memory consumption with a moderate increase of computational intensity. A lower hitrate is acceptable if the overall performance improves. This tradeoff is supported by present hardware trends of increasing memory volumes and low read latencies.

Random write I/O performance and endurance issues make it important to perform sequential or semi-sequential writes. Our initial experiments indicate that when a set of random write I/O requests is concentrated on a relatively small region then they are performed nearly as fast as sequential requests. This effect is also studied in [13]. This obviously lacks the write reduction of the SIAS approach and introduces in-place updates, yet FBARC improves on existing buffer management strategies. This observation affects the way spatial locality can be addressed, especially the choice of the clustering algorithm (Section 7.2.1).

By static clustering the set of dirty pages, the pages for eviction can be subdivided into smaller regions; even though those may not be optimally filled and may result in random writes, they

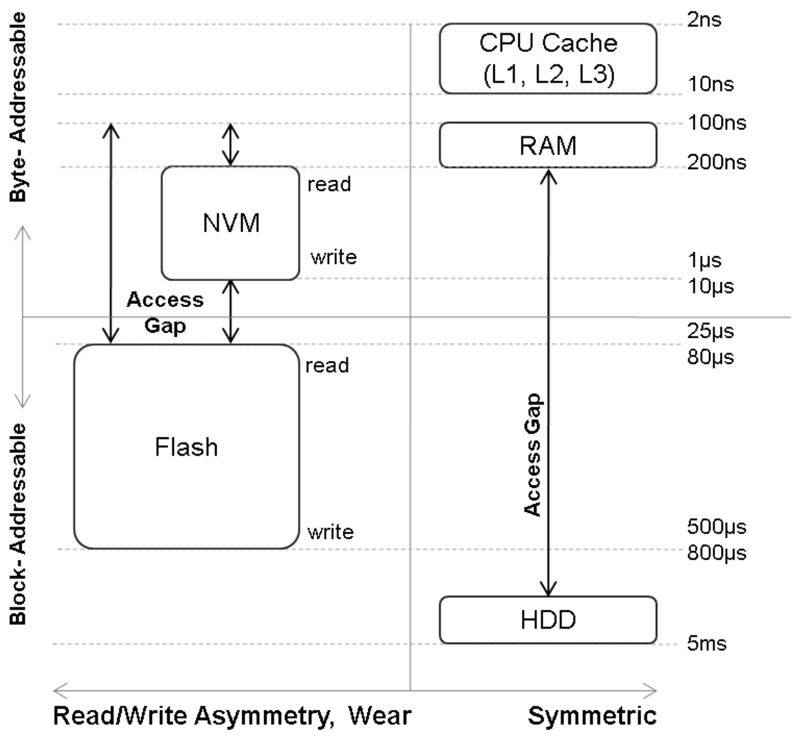


Figure 7.1: Memory Hierarchy and Access Gap. Asymmetric storage technologies fill in the access gap between symmetric storage, questioning the assumption of traditional buffer managers that optimize for hitrate to minimize disk accesses of whatever type (read-/write). The access gap shrinks with asymmetric storage technologies but becomes operation dependent.

will be executed as sequential ones. Hence the desired tradeoff may be achieved at acceptable computational costs.

In the following the FBARC algorithm is explained in more detail.

7.2.1 FBARC - Algorithm

FBARC is an extension of ARC which is explained first in the following paragraph. The organization of the buffer space is depicted in Figure 7.2.

ARC

ARC [77] organizes recently accessed pages in an LRU organized list L1; and frequently accessed pages in another list L2. Both lists L1 and L2 are subdivided into two further lists - L1 into T1 and B1; L2 into T2 and B2. The T-lists (T1 and T2) manage the available buffer frames; the sum of their sizes equals the *total number of buffer frames available* - *c*. One frame has the capacity to store one buffer page.

The B-lists (B1 and B2) are virtual lists, containing metadata about pages which were evicted from the respective T-lists. The B-lists contain in total another - *c* page metadata entries; thus the total page metadata maintained by ARC is *2c* entries.

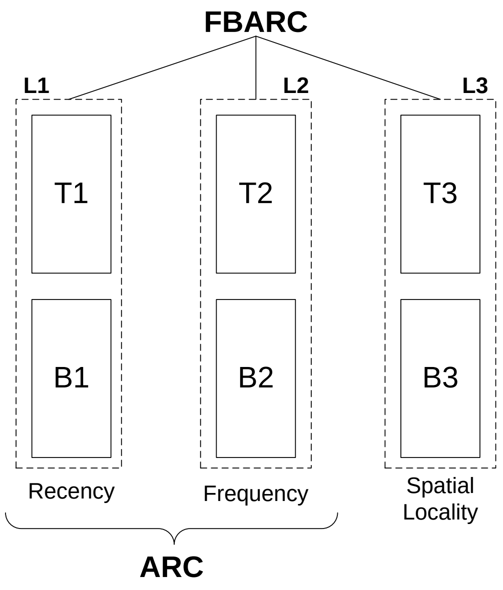


Figure 7.2: List organization of FBARC. L1 and L2 are inherited by ARC [77]. FBARC adds the additional List L3 which is divided into T3 and B3. T3 manages clusters of dirty pages, each containing (a) a list of dirty pages and (b) a reference counter. B3 is LRU-organized and maintains metadata of uncached dirty pages.

ARC balances the sizes of L1 and L2 to realize the trade-off between recency and frequency, which enables adaptation to workload changes [77].

FBARC

FBARC extends ARC with an additional list (L3) to address spatial locality (Figure 7.2). Analogously to L1 and L2 it is subdivided into T3 and B3. Unlike T1 and T2, T3 manages *clusters of dirty pages*. A cluster contains a list of pages and a reference counter. B3 is LRU-organized and contains the metadata for single uncached pages.

FBARC uses a static clustering algorithm (called *GRID clustering*). We logically divide the disk space into static equally-sized regions called clusters.

Even though the clusters can be sparsely filled with dirty pages, which will result in random write requests, they will be performed as if they were sequential (semi-sequential) since the page addresses are concentrated on a relatively small region (cluster).

The cluster size is a tunable parameter, specific to the concrete Flash device. The proposed *GRID clustering* represents a trade-off between efficiency, simplicity and acceptable computational intensiveness.

Assume that the cache is completely filled and a page is requested, this means that some cache has to be freed (one or more pages have to be evicted). First, FBARC verifies whether the page has already been processed by locating the metadata for that page (Algorithm 6, line 2) in L1, L2 or L3. If it finds it, it checks if the page is cached (Algorithm 6, line 3). If so, the page is moved to the MRU position of T2 and the page is returned. If the page was not cached, but the metadata was found, then FBARC uses this information to adjust the balance between T1, T2 and T3 (Algorithm 6, lines 6-16) and then invokes the eviction process (Algorithm 6, lines 17, 18). The page is then placed at the MRU position of T2 and the page is returned (Algorithm 6, line 19).

How the balance between T1, T2 and T3 is changed depends on where the metadata hit occurred.

The lists are managed using a *target size*, which is a soft upper bound under which the lists should stay. If a list is bigger than its target size then the next eviction victim will be chosen from that list.

If the metadata was located in B1, then the target size for T1 is increased by one, if it was in B3 then the target size for T3 is increased by one. If it was in B2 then the logical target size for T2 is increased by one. T2 has only a logical target size, as it is the result of $c - ts_1 - ts_3$. So if the target size of T2 has to grow, another target size has to shrink. To decide which target size has to shrink we compare them with each other (Algorithm 6, line 9), and then shrink the bigger one. This protects the balance in cases where many hits in B2 would otherwise lead to skew and the predominance of the list.

If the page is unknown (e.g. the look up did not return any metadata) then FBARC might have to evict an additional B-list page. This housekeeping ensures that the complete historic view stays confined within $3c$, i.e. information about evicted pages stored in the B-lists.

First, FBARC checks if L1 (i.e. T1+B1) is the same size as the cache size and whether there is at least one entry in B1 (Algorithm 6, line 25). If this is the case then the LRU entry of B1 is removed. Otherwise, it goes on to check whether the total size of all lists (L1, L2 and L3) exceeds the cache size (Algorithm 6, line 27). It then checks if L1+L2 is bigger than or equal to $2c$ (Algorithm 6, line 28). If so, the LRU entry of B2 is removed, otherwise FBARC checks if the total size of all lists (L1+L2+L3) is equal to $3c$ and in that case removes the LRU entry of B3 (Algorithm 6, line 30).

This confines the maximum amount of information about the entries in L1 to c, the maximum amount of entries in L2 to 2c and the maximum size of L3 to 3c. The reasoning behind this is that L1 represents the recency and an excessively large recency list would contain pages that are not recent any more. L2 on the other hand represents frequency, and the information that a page was frequently accessed in the past is useful even if it was not that recent.

Similarly, the information that a page was written to in the past is even more useful, as we try to retain dirty pages longer to reduce the amount of write I/Os that a buffer management strategy produces.

After the housekeeping is done, the eviction process is invoked (Algorithm 6, line 34). The new page is read in, placed at the MRU position of T1 and returned (Algorithm 6, lines 36-39).

Algorithm 6 FBARC: Get Procedure

```
1: procedure GET($x$)
2:     if $x \in buffers$ then
3:         if $x.origin \in \{t_1, t_2, t_3\}$ then                         ▷ Cache Hit
4:             MOVETOMRU($t_2$, $x$)
5:         else                                                 ▷ Cache Miss, Hit in History
6:             if $x.origin = b_1$ then
7:                 $ts_1 \leftarrow$ MAX($c$, $ts_1 + 1$)
8:             else if $x.origin = b_2$ then
9:                 if $|ts_1| \geq |ts_3|$ then
10:                    $ts_1 \leftarrow$ MIN(0, $ts_1 - 1$)
11:                else
12:                    $ts_3 \leftarrow$ MIN(0, $ts_3 - 1$)
13:                end if
14:            else if $x.origin = b_3$ then
15:                $ts_3 \leftarrow$ MAX($c$, $ts_3 + 1$)
16:            end if
17:            EVICT($x$)
18:            $buffers[x] \leftarrow$ READIN($x$)
19:            MOVETOMRU($t_2$, $x$)
20:        end if
21:    else                                                          ▷ Page is unknown
22:        if $|t_1| = c$ then
23:            EVICTLRU($t_1$)                                  ▷ instead of calling evict
24:        else                                                           ▷ Housekeeping
25:            if $|l_1| = c$ then
26:                REMOVELRU($b_1$)
27:            else if $|l_1| + |l_2| + |l_3| \geq c$ then
28:                if $|l_1| + |l_2| \geq 2c$ then
29:                    REMOVELRU($b_2$)
30:                else if $|l_1| + |l_2| + |l_3| = 3c$ then
31:                    REMOVELRU($b_3$)
32:                end if
33:            end if
34:            EVICT($x$)
35:        end if
36:        $buffers[x] \leftarrow$ READIN($x$)
37:        MOVETOMRU($t_1$, $x$)
38:    end if
39:    return $buffers[x]$
40: end procedure
```

The eviction process first checks if a page has to be evicted at all (Algorithm 7, line 2), and if so FBARC continuously evicts as many pages from different source candidates (lists) as needed. At least one buffer slot (size of a page) has to be freed. The source selection for eviction candidate is made based on the current and target size of all lists, and on the metadata (if present) of the requested page. First, it checks if T1 is too big (Algorithm 7, line 3), then it checks if T3 is too big (Algorithm 7, line 5), and if this is also not the case, then it just selects the biggest of all three as the source.

Algorithm 7 FBARC: Eviction

```
1:  procedure EVICT($x$)
2:      while $|buffers| = c$ do
3:          if $|t_1| > 0 \land (|t_1| > ts_1 \lor (x \in b_2 \land |t_1| = ts_1))$ then
4:              $source \leftarrow t_1$
5:          else if $|t_3| > 0 \land (|t_3| > ts_3 \lor (x \in b_2 \land |t_3| = ts_3))$ then
6:              $source \leftarrow t_3$
7:          else
8:              $source \leftarrow$ MAX($t_1, t_2, t_3$)
9:          end if
10:         EVICTLRU($source$,$x$)
11:     end while
12: end procedure
```

If the eviction source candidate is T1 or T2 then FBARC attempts to evict the page at the LRU position (Algorithm 8, line 5). If that page is clean, then the buffer for it is cleaned, and its metadata is moved to the corresponding bottom list (Algorithm 8, lines 9, 10). If it is dirty it will be added to T3 (Algorithm 9) and the eviction process is restarted (Algorithm 7, line 2) as the dirty page was not written out, and thus no free place was created yet. If the source is T3, then a special eviction is started (Algorithm 10).

Algorithm 8 FBARC: Eviction of LRU for l_1 and l_2

```
1:  procedure EVICTLRU($source, x$)
2:      if $source = t_3$ then
3:          EVICTFROMT3($nil$)
4:      else
5:          $lru \leftarrow$ REMOVELRU($source$)
6:          if $lru.dirty$ then
7:              ADDTOT3($lru$)
8:          else
9:              REMOVE($buffers[lru]$)
10:             MOVETOMRU($source.history, lru$)
11:         end if
12:     end if
13: end procedure
```

When a page is added to T3, FBARC calculates its cluster id based on the page number and a given cluster size (Algorithm 9, line 2). If a cluster with that cluster id already exists, then the page is added to it and its reference counter is incremented (Algorithm 9, lines 10, 5). If there is no pre-existing cluster with that id, then a new one is created and added to the cluster hash map (Algorithm 9, lines 7, 8).

Algorithm 9 FBARC: Add Page to t_3

```
 1: procedure AddToT3($x$)
 2:     $clusterId \leftarrow page/clusterSize$
 3:     if $clusterId \in clusters$ then
 4:         $cluster \leftarrow clusters[clusterId]$
 5:         $x.cluster.refCount++$
 6:     else
 7:         $cluster \leftarrow$ new Cluster()
 8:         $cluster[clusterId] = cluster$
 9:     end if
10:     $cluster.pages.$append($x$)
11: end procedure
```

There are two reasons for a page to leave T3. The first is that the page was hit by an access and is moved to T2 (Algorithm 6, line 4). In that case FBARC removes the page from its cluster and increases its reference count by one (Algorithm 10, lines 2, 3, 7). The reasoning behind this decision is that the page might come back into the cluster if it stays long enough. If the cluster is empty after this operation, then the cluster is removed (Algorithm 10, lines 4, 5). The second reason is that T3 was selected as a source in the eviction process. In that case FBARC estimates the eviction cost for each cluster ($clusterCost$) and selects the cluster with the lowest cost as the victim (Algorithm 10, lines 10, 11).

The cost is estimated by the reference count and the amount of pages that are in the cluster. Thus a cluster with a low reference count and a lot of pages is more likely to be replaced then a cluster with the same reference count and only a few pages. This allows a cluster with only a few pages to live longer and thus gives it a chance to accumulate more pages. A selection solely based on the cluster size is insufficient, since then very small clusters could block a buffer slot for a long time.

After the victim cluster is chosen, the reference count of all other clusters is decreased by the victims reference count (Algorithm 10, line 13). This is mainly to keep the reference count low such that inactive clusters that once had a high activity can be evicted sooner. The pages of the victim cluster are then written in sequential order to the storage and added to the MRU position of B3.

7.3 FBARC: Instrumentation and Experiments

We experimentally evaluated the performance of FBARC, comparing it against a number of different buffer management algorithms (ARC, FOR+, LRU, CFLRU, CFDC) under realistic workloads (TPC-C, TPC-H, pgbench) in different scenarios. We implemented a simulation framework, hosting and executing the set of buffer management strategies. An overview of the trace-driven simulation process is provided in Figure 7.3. It comprises three phases, which are described in detail in the following Sections:

- (i) Recording raw traces for different DB workloads (Section 7.3.1, and Section 7.3.2);

Algorithm 10 FBARC: Evict Page from t_3

```
1: procedure EvictFromT3($x$)
2:     if $x \neq nil$ then
3:         $x.cluster.pages$.remove($x$)
4:         if $|x.cluster.pages| = 0$ then
5:             $clusters$.remove($x.cluster$)
6:         else
7:             $x.cluster.refCount++$
8:         end if
9:     else
10:        $clusterCost \leftarrow \lambda x \leftarrow (x.refCount, -|x.pages|)$
11:        $bestCluster \leftarrow$min($clusters$, $key = clusterCost$)
12:        for all $cluster \in clusters$ do
13:            $cluster.refCount -= bestCluster.refCount$
14:        end for
15:        $bestCluster.pages$.sort()
16:        for all $page \in bestCluster$ do
17:            writeOut($page$)
18:            MoveToMru($b_3$,$page$)
19:        end for
20:        $clusters$.remove($bestCluster$)
21:    end if
22: end procedure
```

- (ii) simulation of the eviction policies and generation of block access traces (Section 7.3.2);
- (iii) execution of the I/O traces on a physical SSD using FIO (Section 7.3.2).

With this setup we are able to study:

- different aspects of spatial and temporal locality, e.g. hitrate and quality of the clustering algorithm;
- the computational intensiveness and the I/O behavior;
- various additional aspects of the buffer management algorithms, e.g. scan-resistance or workload adaptation.
- In addition this setup provides a controlled environment and enables the evaluation under repeatable conditions and realistic workloads. For fair performance evaluation we assign equal total amount of memory to all compared algorithms (unless explicitly stated otherwise).

The following system was used to perform the experimental evaluation: 4 GB RAM, Intel Core 2 Duo 3 GHz running under Linux (kernel 2.6.41). In addition, we used an Intel X25-E/64GB enterprise SSD and a Hitachi HDS72161 7200RPM SATA2 320GB HDD; the latencies of both drives are listed in Table 7.1.

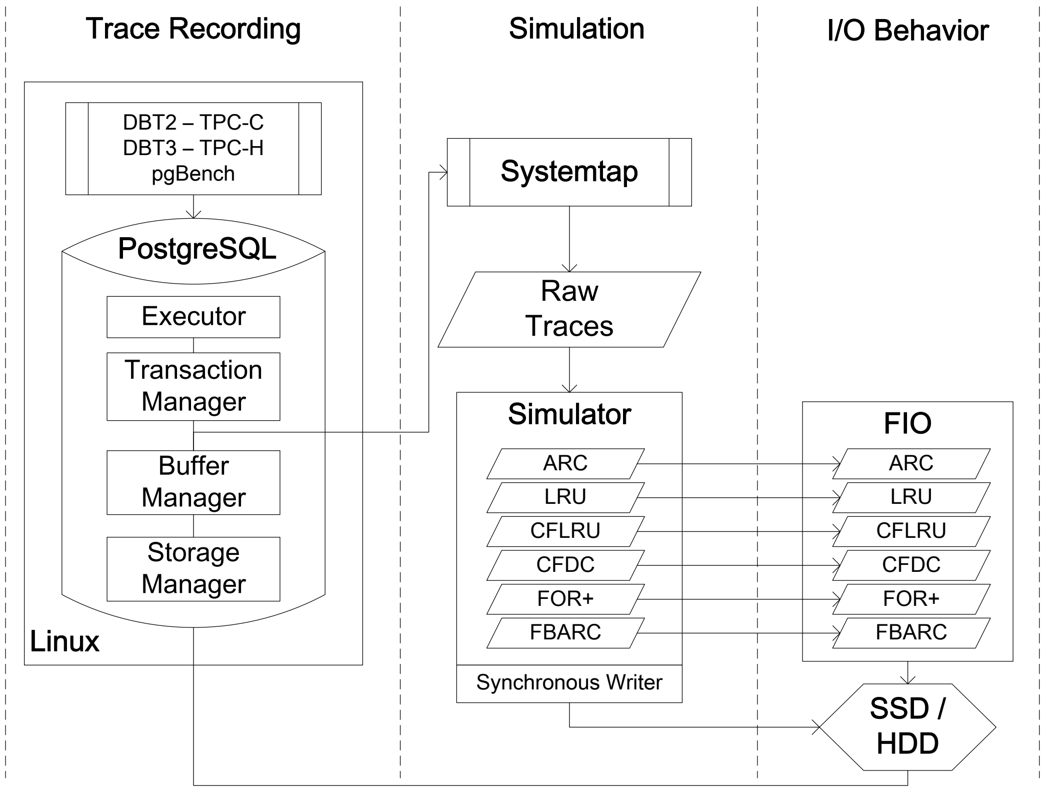


Figure 7.3: Simulation Process. (i) Trace Recording: raw traces are recorded by setting appropriate probing points; (ii) Simulation: simulation of the eviction strategies and genertion of block I/O traces; (iii) I/O Behavior: block trace execution on physical storage media (SSD and HDD).

Table 7.1: Average I/O Latencies for the Intel X25-E SSD and the Hitachi HDS72161 HDD.

Blocksize 4KB	**SSD Intel X25-E 64GB**	**HDD Hitachi HDS72161**
Sequential Read	53 μs	0,133ms
Sequential Write	59 μs	0,168ms
Random Read	167 μs	10,8ms
Random Write	435 μs	5,6ms (write cache)

7.3.1 Raw Trace

To ensure repeatable experiments and comparable experimental results under realistic workloads we record traces of PostgreSQL buffer manager method calls. Probing points were set on buffer methods *fetching* a page as well as those *marking a buffer dirty* within PostgreSQL's *bufmgr.c*. We used a Fedora Linux (kernel 2.6.41) with the *Systemtap* extension (translator 1.6; driver 0.152) to record the trace. By setting probing points accordingly (prior to the buffer manager, Figure 7.3), we guarantee that the traces record the real database behavior for the respective OLTP or OLAP workload, and eliminate the influence of PostgreSQL's eviction policy and storage manager. These are simulated for each implemented strategy by the simulation framework (Section 7.3.2)

7.3.2 Workload and Trace Characterization

An out of the box PostgreSQL (version 9.1.1) was instrumented for OLTP and OLAP workloads. Both types of workloads are considered since they stress different aspects of a buffer management strategy, and target asymmetry differently.

As an OLTP benchmark we used *DBT2* [22], an open source implementation of the TPC-C benchmark and *pgbench* [24], which is the PostgreSQL internal benchmark. DBT2 was instrumented with 200 Warehouses (nominal DB size approx. 20GB), 10 database connections and 10 terminals per warehouse. PostgreSQL uses 24 MB shared buffer space. The other OLTP trace was recorded with *pgbench*, which was instrumented with a scale factor of 600.

As OLAP load we used *DBT3* [23], an open source implementation of the TPC-H benchmark. PostgreSQL was instrumented with scale factor 3 (nominal DB size approx. 13GB). The default database page size of PostgreSQL (8KB) is used throughout all benchmarks.

For the HDD benchmarks, we used scaled down traces that have the same workload characteristics as the *pgbench* and *DBT2* traces. They are scaled down to one tenth of the original trace size (nominal DB size approx. 2GB). Scaling was performed to reduce the otherwise unacceptably long experimental run-time.

All traces contain index and table accesses. The traces have, however, been purged of PostgreSQL specific *service* accesses (e.g., management overhead) to make them more comparable with traces that would be gained by other MV-DBMSs.

Figures 7.4, 7.5, 7.6 and 7.7 show the access distribution per trace. The dot-dashed blue and dotted green lines show the access frequency per distinct page, while the dashed red and continuous yellow lines show the cumulative part of the total accesses a distinct page receives. We have chosen a logarithmic scale for both the access frequency as well as the distinct page count to emphasize the impact of the most frequently accessed pages.

Trace H

DBT3 (TPC-H) is a typical OLAP benchmark, and that is also reflected in its trace. Figure 7.4 shows the low fraction of write accesses in the TPC-H throughput test (temporary data), hence we use this trace mostly to investigate the effects of the write awareness extensions on a mostly read only trace. Additionally, as depicted in the scatter diagram of this trace, it executes complex queries that focus on only a few relations at a single time (see Figure 7.15), which is reflected by the plateaus seen in Figure 7.4.

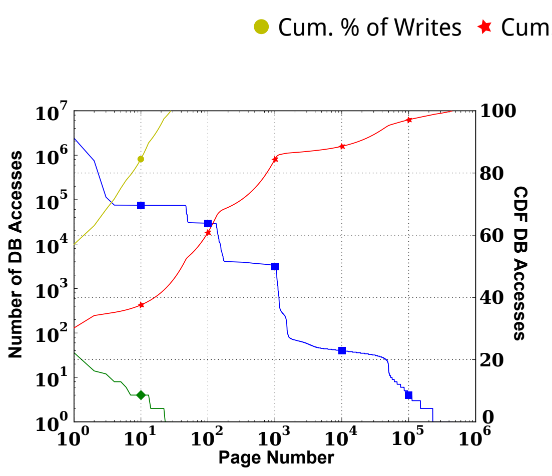


Figure 7.4: Trace H: DBT3 (TPC-H) - Scale Factor 3. Page numbers from hot (left) to cold (right) pages.

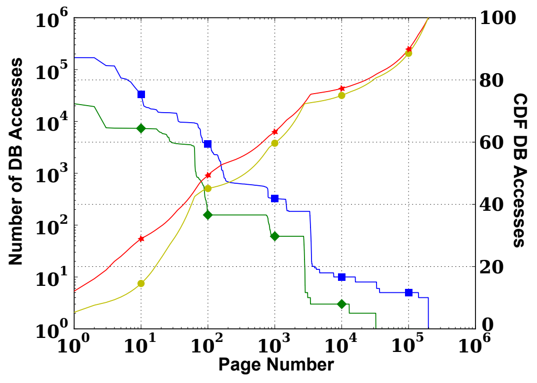


Figure 7.5: Trace B: pgBench - Scale Factor 600. Page numbers from hot (left) to cold (right) pages.

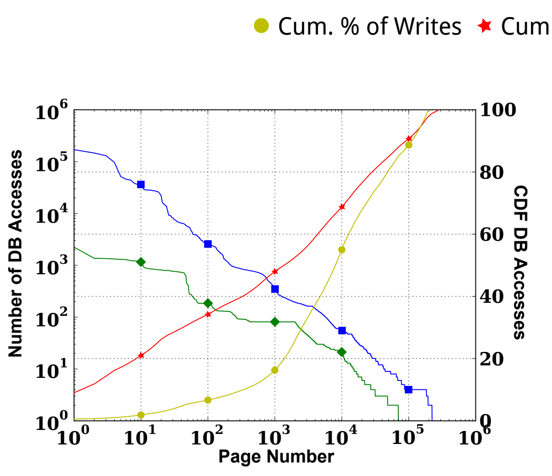


Figure 7.6: Trace C: DBT2 (TPC-C) - 200 Warehouses. Page numbers from hot (left) to cold (right) pages.

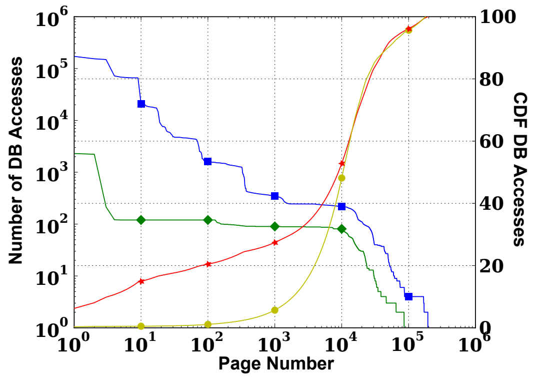


Figure 7.7: Trace C_d: DBT2 with 200 Warehouses. Delivery transaction accesses. Page numbers from hot (left) to cold (right) pages.

Trace B

The pgbench trace shows a TPC-B like access pattern. Even though some relations are accessed more frequent than others, the access distribution within the relations themselves is very uniformly random (Figure 7.18). That can also be observed in Figure 7.5, as every plateau represents a single relation, and all the pages within it are equally often accessed. The writes are almost equally distributed, but they occur ten times less often.

Trace C

For DBT2 we have recorded two traces. One, as shown in Figure 7.6, contains all accesses as they are defined in the TPC-C specification (Trace C). The other, as shown in Figure 7.7 (Trace C_d), contains only accesses from the delivery transaction. While trace C allows us to create a realistic OLTP load, trace C_d allows us to have a write heavy OLTP-like load.

Figure 7.6 and 7.7 show that for both traces more then half of all accesses are in a very narrow range of pages, as less then 10% of all pages account for more then 60% of all accesses. The write requests though are a lot less concentrated in both traces. Additionally the trace C_d shows a pattern of concurrent semi sequential accesses over most of the involved relations (see Figure 7.17).

Trace SR

In order to stress the scan resistance of the different algorithms the framework simulates large sequential accesses by introjecting *parasites* into the OLTP traces. After an amount of OLTP load a parasite is introjected. The parasite pollutes the complete cache size and accesses pages which are not covered by the main trace to avoid any interference. After that the OLTP load is resumed for n additional requests, where n is the cache size. The Load is resumed for only n additional requests to highlight the effect of the parasite. Each parasite is either write only or read only (see Section 7.4.4).

Instrumentation - Simulation and Metrics

The simulation framework applies the buffer eviction policies to the raw traces. The simulated eviction policies are:

- ARC
- LRU
- CFLRU with 10% priority region.
- CFLRU with 40% priority region.
- CFDC with 10%priority region.
- CFDC with 40% priority region.
- FOR+
- FBARC with different cluster sizes.

The simulation results are reported for the following metrics:

- *hitrate*

		Trace B			Trace C			Trace C_d			Trace H		
		1024	2048	4096	1024	2048	4096	1024	2048	4096	1024	2048	4096
Hitrate [%]	LRU	90.26	91.40	92.29	78.68	81.55	83.77	76.53	79.06	79.38	90.02	91.12	92.63
	ARC	89.91	91.29	92.33	78.60	81.06	83.24	76.54	78.98	79.09	89.78	91.07	92.53
	FBARC	88.39	90.43	92.11	77.68	81.17	83.82	73.02	78.40	79.40	89.88	91.06	92.46
	CFLRU 10%	90.13	91.24	92.20	78.48	81.40	83.60	76.19	79.08	79.38	89.92	91.07	92.57
	CFLRU 40%	89.69	90.63	91.73	77.25	80.64	82.88	74.02	78.56	79.42	89.44	90.81	92.40
	CFDC 10%	90.10	91.24	92.20	78.15	81.21	83.50	75.67	79.06	79.39	89.90	91.05	92.57
	CFDC 40%	89.46	90.58	91.72	75.82	79.74	82.34	72.20	78.18	79.66	89.34	90.72	92.38
	FOR+	90.24	91.31	92.25	78.38	81.53	83.49	74.66	78.91	79.62	89.78	90.94	92.52
CPU [s]	LRU	55.94	58.61	63.79	70.74	79.91	94.36	125.83	139.92	145.03	147.07	153.36	179.49
	ARC	63.29	68.60	71.95	78.96	87.70	104.10	138.88	144.62	155.76	167.23	182.80	201.99
	FBARC	81.79	84.18	96.54	152.08	177.51	199.36	293.10	333.55	317.33	188.06	195.14	213.23
	CFLRU 10%	70.43	86.24	113.82	102.13	135.66	195.97	189.31	248.65	371.63	226.18	293.91	414.02
	CFLRU 40%	111.30	160.82	254.13	209.25	315.38	533.11	391.34	593.98	1038.69	223.49	288.71	413.21
	CFDC 10%	80.80	100.34	139.46	113.43	145.13	206.45	190.77	248.10	323.42	163.69	167.02	194.83
	CFDC 40%	130.86	196.65	331.69	198.89	298.83	501.99	328.95	487.48	777.72	161.15	165.83	176.39
	FOR+	174.57	218.34	282.86	257.75	365.33	545.26	352.24	442.31	694.68	392.49	546.64	938.95
I/O [s]	LRU	166.23	156.69	148.07	286.43	250.84	237.59	539.22	484.94	477.87	267.94	257.12	217.49
	ARC	167.55	158.43	148.52	297.54	258.42	232.95	536.95	486.32	485.71	264.70	232.34	215.56
	FBARC	180.44	164.19	148.93	299.94	255.12	224.01	581.28	478.80	441.83	263.56	232.03	216.34
	CFLRU 10%	164.90	157.92	150.86	295.90	254.19	232.62	539.56	485.13	477.74	268.76	256.51	217.32
	CFLRU 40%	171.79	162.42	153.66	297.26	257.29	233.63	566.00	492.66	476.41	269.35	256.42	217.49
	CFDC 10%	167.52	158.64	150.75	301.62	257.59	230.59	554.14	488.28	476.08	269.22	256.74	217.77
	CFDC 40%	173.26	161.02	154.88	329.66	280.03	247.45	613.08	504.01	471.26	268.43	255.33	217.06
	FOR+	164.33	155.48	149.41	292.92	255.91	232.48	559.94	490.47	474.07	268.73	256.54	216.55
Combined[s]	LRU	175.61	166.45	160.18	324.99	288.70	259.57	570.74	516.32	505.07	277.59	265.47	267.89
	ARC	180.16	167.28	159.38	327.83	293.85	264.74	571.20	518.18	513.92	275.08	273.24	284.83
	FBARC	191.48	172.92	160.94	324.19	278.64	245.99	607.10	495.23	456.05	278.14	278.72	292.41
	CFLRU 10%	177.84	167.41	156.95	322.06	285.56	254.51	567.89	509.35	494.43	290.39	321.26	426.94
	CFLRU 40%	179.41	179.68	261.75	326.04	340.06	535.93	583.95	620.08	1048.52	290.96	324.15	423.25
	CFDC 10%	176.43	164.38	161.37	328.39	291.04	254.63	585.46	511.57	497.51	278.87	267.01	280.63
	CFDC 40%	180.71	212.96	341.38	354.38	324.54	512.10	641.94	537.00	834.27	278.66	266.79	262.18
	FOR+	194.31	235.83	294.65	315.24	388.96	565.95	584.18	510.23	723.91	454.84	599.22	971.54

Figure 7.8: Results for Hitrate, CPU, I/O and Overall Time. Columns indicate amount of pages in the cache (1024 .. 4096). Charts covering specific aspects are presented in Figure 7.8, 7.11, and 7.14.

- *CPU time* - measure of computational complexity (Section 7.4.1).
- *I/O time* - measure of I/O performance (Section 7.4.2).
- *overall runtime* - interleaving of CPU and I/O time (Section 7.4.3).

I/O Trace

In the final step we benchmarked the traces on a physical SSD and HDD. The simulation framework translates page addresses into block addresses generating an I/O trace, which is used as an input for the FIO benchmark. FIO accesses the SSD/HDD as a raw device using direct I/O, and thus bypasses the I/O cache. We used FIO for the following reasons: a) guaranteed I/O behavior; b) controlled use of I/O parallelism; c) reliable performance metrics (IOPS, runtime). On SSD after each run of a single trace, we executed a series of random write operations (approximately 30GB) using FIO to avoid SSD state dependencies.

7.4 FBARC: Experimental Evaluation

The experiments of our experimental evaluation include: (i) investigation of the hitrate and the computational intensiveness of all algorithms (Section 7.4.1); (ii) examination of the I/O behavior of all buffer management strategies (Section 7.4.2); as well as (iii) the investigation of the overall time (Section 7.4.3). We investigate the influence of sequential operations on the algorithms (scan resistance), which gains importance in database systems optimized for new storage technologies or analytical processing (Section 7.4.4). Finally, we compare the performance on HDDs to investigate whether the algorithms have desirable properties on a variety of storage media. Since all those experiments vary the cache size but keep the FBARC specific parameters, i.e. cluster size constant, we additionally analyze the influence of the FBARC parameters in terms of the influence of the cluster size on the computational complexity and I/O behavior (Section 7.4.5). CFLRU and CFDC in particular are analyzed with two different priority queue sizes of 10% and 40%. All results presented in this section are the average values of three experimental runs (since the standard deviation was consistently low we do not report it).

We have consolidated the results of all main experiments into Figure 7.8. This allows to easily compare the results of different traces and see the influence of different factors on the combined metric. The algorithms are sorted in the given order, to allow for an easier comparison of FBARC with ARC and LRU.

7.4.1 CPU Time

In this section we examine the computational complexity (CPU time) and hitrate of each eviction policy, for different traces and buffer/cache sizes (Figure 7.8, rows 9-16, “CPU”). *CPU time* measures the runtime required for each algorithm to complete the respective trace under full CPU utilization. It is an indication for the algorithm’s pure *computational complexity* on the respective trace. I/O time and parallelism are not present.

Here LRU naturally outperforms all other algorithms. With increasing cache size FBARC has better runtime than the others.

For small cache sizes FBARC performs mostly comparable to or slower than the other strategies, but as the cache size increases that gap narrows. While the hitrates for the different algorithms are comparable and grow with the amount of cache used, the CPU times differ significantly. The trend to larger cache sizes is in favor of the FBARC algorithm.

FOR+ exhibits the highest computational complexity of all algorithms. It increases significantly with growing cache sizes. CFDC exhibits the highest computational intensiveness of all clustering algorithms especially for larger cache sizes. This is due to the employed clustering algorithm. CFLRU and FBARC have similar complexities, with CFLRU being better for smaller buffer sizes and FBARC taking lead for larger buffer sizes. LRU and ARC are both basic algorithms and thus shine in this metric because they are write-oblivious and do not have the respective complexity. Using the computational resources of modern many core CPUs for the buffer eviction strategy is valid, since most of the computational power is unused.

Upcoming NVM technologies are characterised by higher asymmetry, smaller page sizes and hence yield higher CPU overhead for T3 Maintenance. The relative CPU overhead of FBARC compared to other strategies is (Figure 7.9): (i) smaller; (ii) it also grows slower for increasing number of pages. In general, all clustering strategies are affected; to attack the issue the ample computational resources of many core CPUs need to be used.

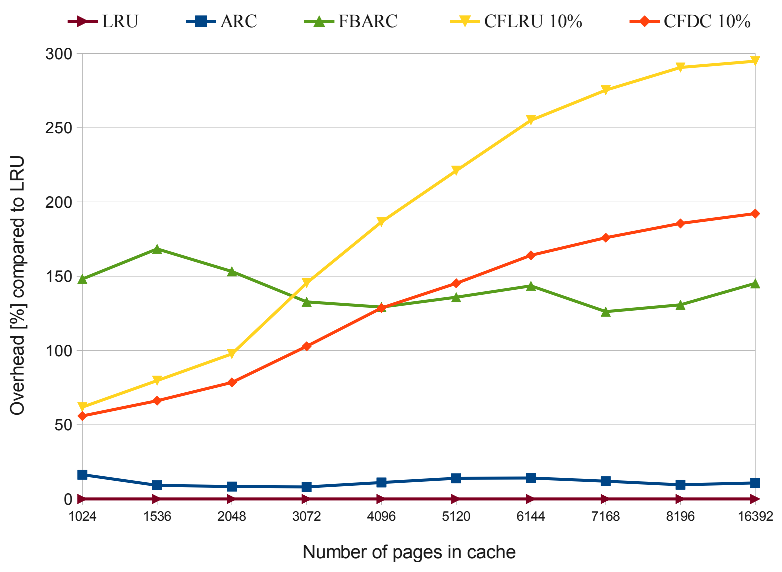


Figure 7.9: Relative CPU overhead for varying number of pages in cache. Baseline LRU. CPU overhead increases for CFLRU and CFDC with a higher amount of pages. FBARC experiences more overhead than ARC due to the GRID clustering algorithm.

Table 7.2: Read KB in Trace C. Columns indicate amount of pages in the cache (1024 .. 4096).

Algorithm	1024 pages	2048 pages	4096 pages
LRU	6,485,952	5,613,128	4,938,152
ARC	6,511,440	5,761,776	5,098,056
FBARC	6,790,968	5,728,960	4,920,816
CFLRU 10%	6,549,112	5,657,208	4,988,896
CFLRU 40%	6,922,672	5,888,768	5,208,096
CFDC 10%	6,648,344	5,716,696	5,019,144
CFDC 40%	7,358,888	6,163,464	5,373,144
FOR+	6,580,288	5,618,288	5,021,456

7.4.2 I-O Time

In this section we report the I/O times for the investigated replacement strategies for different loads and buffer cache sizes. *I/O time* stands for the I/O time needed to write/read evicted/requested pages for the respective raw (benchmark) trace. The framework creates an I/O trace that corresponds to the simulated eviction policy and is used as input for FIO (Section 7.3.2). It is then executed with synchronous direct I/O. The CPU time is minimal and can therefore be neglected.

FBARC reads and writes faster than the other eviction policies. The higher write rate of FBARC is the result of the sequentialization of random writes. It is significantly faster to write a whole cluster of pages which reside in spatial locality than writing random.

The results also show that the preferred eviction of clean pages leads to a higher value of read I/O (Table 7.2) and a higher read IOPS, nevertheless the influence of the prioritized eviction

Table 7.3: Written KB in trace C. Columns indicate amount of pages in the cache (1024 .. 4096).

Algorithm	1024 pages	2048 pages	4096 pages
LRU	3,168,016	2,908,216	2,691,712
ARC	3,135,872	2,880,960	2,687,672
FBARC	3,209,080	2,911,360	2,665,472
CFLRU 10%	3,123,869	2,878,440	2,674,600
CFLRU 40%	3,028,568	2,800,088	2,632,928
CFDC 10%	3,210,520	2,933,828	2,703,720
CFDC 40%	3,393,264	3,039,200	2,788,552
FOR+	2,965,504	2,717,352	2,595,992

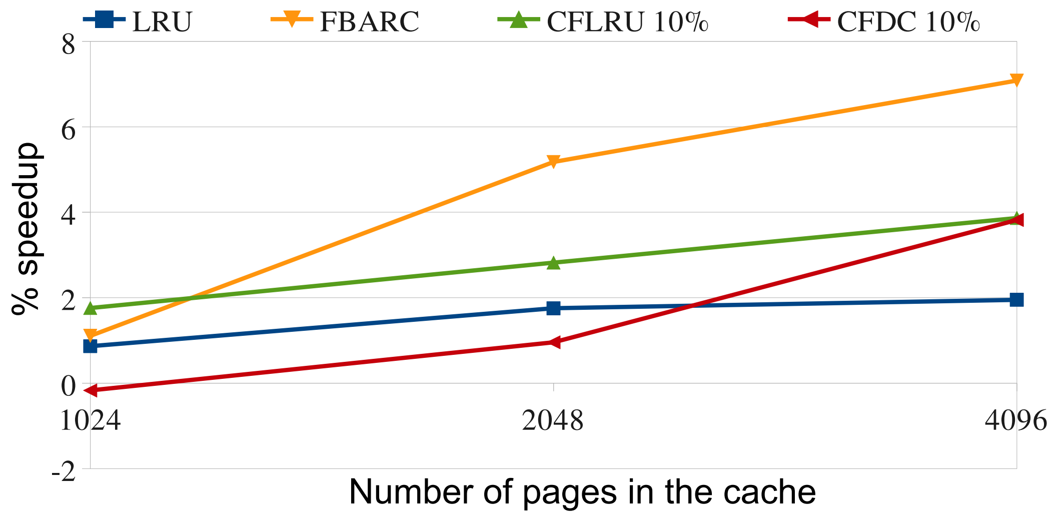


Figure 7.10: Overall speedup in percent compared to ARC on trace C

of clean data pages does not impact the hitrate negatively, as the results of the CPU time tests show.

FOR+ shows the lowest amount of written bytes (Table 7.3). This is the result of its meta data use. It retains single hot dirty pages longer in cache than any other algorithm, due to its comparatively long write history list. Even though FOR+ writes less than FBARC it is not faster per se since the resulting write patterns lack spatial locality.

7.4.3 Overall Time

The *overall time* represents a combined metric unifying the *CPU time* and *I/O time*. It interleaves the computational tasks of a buffer strategy with I/O time for page fetch/eviction (Figure 7.8, rows 25-32, *Combined*). The I/O requests are executed as synchronous I/O: its blocking nature guarantees that the complete I/O latency is accounted for. Thus I/O parallelism and asynchronous I/O are not present which represents the realistic worst case scenario. (In commercial implementations I/O parallelism will be present to varying degrees, increasing the performance.)

Figure 7.8 (rows 25-32, *Combined*) depicts the *overall time* of the algorithms under test and Figure 7.10 and 7.11 depict the relative improvement over ARC on trace C and C_d. We removed

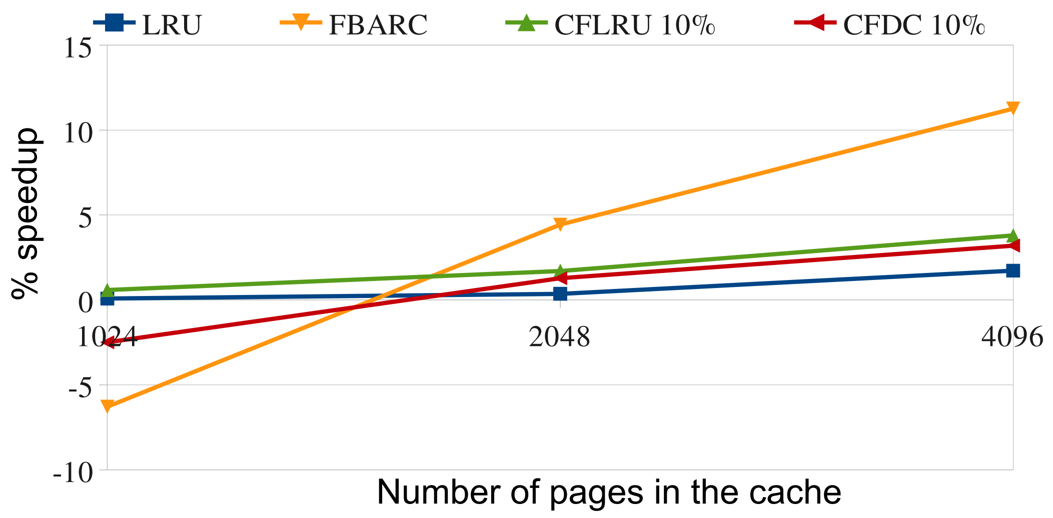


Figure 7.11: Overall speedup in percent compared to ARC on trace C_d.

CFDC 40%, CFLRU 40% and FOR+ from those charts, because of their significant computational overhead (see also Figure 7.8 (rows 13, 15, 16, *CPU*)).

FBARC works very well on trace C and gains up to 7% speedup with a big cache size. On trace C_d FBARC achieves a speedup of up to 11% but slows down with a smaller cache size. The performance slowdown comes mainly from a comparatively low hitrate for that particular cache size. Even though FBARC has a higher computational intensiveness than some of the other strategies, its resulting I/O behavior does reduce the *overall time*.

In trace H there were so few write requests, that the hitrate and computational overhead are the main source of differences between the different strategies.

In trace B the uniform random access seems to produce equally mostly unchanged results when compared to LRU and ARC, thus none of the write aware strategies can really improve over the basic strategies there.

7.4.4 Scan Resistance

Scan resistance is an important characteristic of buffer management algorithms with view of new storage technologies and analytical loads and cache-awareness. At their core both trends employ sequentializing write or read accesses, and become increasingly wide-spread. Not every buffer management algorithm is able to handle such type of accesses efficiently.

We compare the hitrate of an eviction policy on a trace without and with (a) a *read parasite* and (b) a *write parasite* introjected. Our findings show that the smaller the cache size the more likely the hitrate will drop on all strategies but ARC and FBARC. FBARC is influenced by neither write nor read parasites, while the other policies are significantly influenced. FBARC, like ARC, uses the recency list to filter out pages that are accessed only once, and thus a sequential parasite can only pollute a relatively small part of the cache. Even though a sequential write parasite can flush the T3 list, it does not influence the hitrate in a significant way as these pages are not frequently accessed. Additionally it will buffer the write parasite and write it out sequentially.

CFLRU, CFDC and LRU lose more than 10% of hitrate on smaller cache sizes while FOR+ looses a quarter of that. LRU derivatives such as CFDC and CFLRU react differently to read and write parasites due to the specific optimizations. As shown in Figure 7.12 the effect of

Hitrate [%]	128	256	512	1024	2048	128	256	512	1024	2048
ARC	87.89	90.23	91.70	92.92	93.12					
ARC (Read)	87.89	90.63	91.70	92.92	93.09	0.00	0.39	0.00	0.00	-0.02
ARC (Write)	87.89	90.63	91.70	92.92	93.09	0.00	0.39	0.00	0.00	-0.02
CFDC	88.67	90.63	91.80	92.77	93.09					
CFDC (Read)	80.08	83.20	85.94	88.62	90.14	8.59	7.42	5.86	4.15	2.95
CFDC (Write)	76.17	80.27	83.40	86.47	88.18	-12.50	-10.35	-8.40	-6.30	-4.91
CFLRU	88.67	90.63	91.80	92.77	93.12					
CFLRU (Read)	81.25	84.38	87.89	89.84	90.14	-8.20	-6.25	-3.91	-2.98	-3.00
CFLRU (Write)	76.17	80.27	83.40	86.47	88.18	-13.28	-10.35	-8.40	-6.35	-4.96
FBARC	88.28	90.43	91.70	92.92	93.04					
FBARC (Read)	87.89	90.43	91.60	92.92	92.92	-0.39	0.00	-0.10	0.00	-0.12
FBARC (Write)	88.28	90.43	91.70	92.92	92.94	0.00	0.00	0.00	0.00	-0.10
FOR+	89.45	90.63	91.80	92.82	93.14					
FOR+ (Read)	84.77	87.50	88.67	89.45	89.77	-4.69	-3.13	-3.13	-3.37	-3.37
FOR+ (Write)	85.55	87.50	88.67	89.45	89.79	-3.91	-3.13	-3.13	-3.37	-3.34
LRU	88.67	90.63	91.80	92.97	93.12					
LRU (Read)	76.17	80.27	83.40	86.47	88.18	-12.50	-10.35	-8.40	-6.49	-4.93
LRU (Write)	76.17	80.27	83.40	86.47	88.18	-12.50	-10.35	-8.40	-6.49	-4.93

Figure 7.12: Influence of Scan Resistance. Absolute Hitrate on the left, difference compared to baseline on the right. Columns indicate amount of pages in the cache (128 .. 2048).

Table 7.4: FBARC - Trace C: influence of cluster size on (a) hitrate, (b) CPU time, (c) overall time. Columns indicate the cluster size.

FBARC - Trace C	1024 pages	2048 pages	4096 pages
Hitrate	83,82	83,83	83,85
CPU time	201,02	182,57	152,56
Overall time	243,88	238,61	230,79

a write parasite on hitrate is twice the effect of a read parasite. LRU does not distinguish between reads and writes hence the identical numbers. The identical hitrate reduction for write parasites for CFDC and CFLRU can be easily explained with the fact that the parasite pollutes the whole buffer cache, regardless of the clustering algorithm and write optimization. The effect of read parasites on hitrate reduction of CFDC and CLRU depends on the cache size. For smaller cache sizes CFLRU exhibits smaller hitrate penalties, a fact that is due to the way the balance between read and write regions (Working/Clean-first region in CFLRU vs. Working/Priority region in CFDC) is being handled. FOR+ is not scan resistant, it however shows smaller hitrate reductions than LRU, since it uses additional information to keep hot pages longer than previously unknown pages. As for FOR+, regardless of all optimizations read and write parasites affect its hitrate reduction to the same degree.

7.4.5 FBARC Cluster Sizes

Throughout all the cluster size experiments we kept the cluster size of FBARC constant. Since it accounts for physical parameters of the Flash Device it is a tunable parameter in FBARC. In this experiment we vary it from 1024 pages to 4096 pages, keeping the buffer size constant at 4096 pages.

	128	256	512	1024	2048
LRU	4720.22	3959.84	3580.07	3219.36	2880.25
ARC	4691.32	4005.01	3625.47	3259.75	2900.58
FBARC	4740.55	3964.56	3568.04	3210.92	2863.26
CFLRU 10%	4762.38	3968.51	3578.98	3231.58	2895.06
CFLRU 40%	5062.04	4183.72	3665.22	3327.70	2981.58
CFDC 10%	4791.82	3975.31	3572.79	3196.70	2829.28
CFDC 40%	5106.15	4114.93	3568.73	3197.88	2815.65
FOR+	4868.25	4034.39	3553.66	3263.65	2977.73

Figure 7.13: I/O Time [s] on HDD. Columns indicate amount of pages in the cache (128 .. 2048).

Table 7.4 shows that the cluster size has a marginal influence on the hitrate. For most traces the hitrate is unaffected or changes by less than 1%. Given a skewed workload where dirty buffers in L3 (Figure 7.2) are requested will result in a cache hit under FBARC because of the CLOCK based cluster management algorithm.

The cluster size has a significant impact on the overall time (Table 7.4). The pages of the evicted cluster are written out as semi-sequential writes and all I/O operations are synchronous (Algorithm 10), which confirms our initial hypothesis. In addition larger cluster sizes reduce the amount of clusters managed by L3 and hence the computational complexity of FBARC.

7.4.6 Behavior on HDD

In this section we evaluate how the different strategies behave on HDD. It is favorable if a special purpose algorithm performs well even in cases in which it was not intended to be used. For this test we scaled the used traces down to reduce the experimental run-time to an acceptable range. For this reason smaller cache sizes are included in the results.

Figure 7.13 shows the overall time for the scaled down trace C to complete. Figure 7.14 shows the relative speedup or slowdown compared to ARC. Most of the values are within 5% of ARC and most LRU derived strategies perform very similar.

These results also show that the computational intensiveness does not influence the overall time in any significant way. The I/O time and hitrate mostly dominate the overall performance.

Blocktrace Diagrams

Figures 7.15, 7.16, 7.17 and 7.18 show the accesses of each trace. The darker a point, the more often a page at this point was accessed. Each point represents several thousand pages.

7.4.7 FBARC Summary and Conclusion

Low-latency storage technologies with asymmetric performance characteristics call for buffer management strategies that treat both temporal and spatial locality as a first class criterion. Hence replacement strategies need to optimize for both hitrate and eviction cost to ensure balanced performance. FBARC is a buffer management strategy addressing I/O asymmetry on Flash devices. On real world traces FBARC is less computationally-intensive (up to 2.5x) due to the employed Grid clustering algorithm. On the traces used, FBARC exhibited the lowest

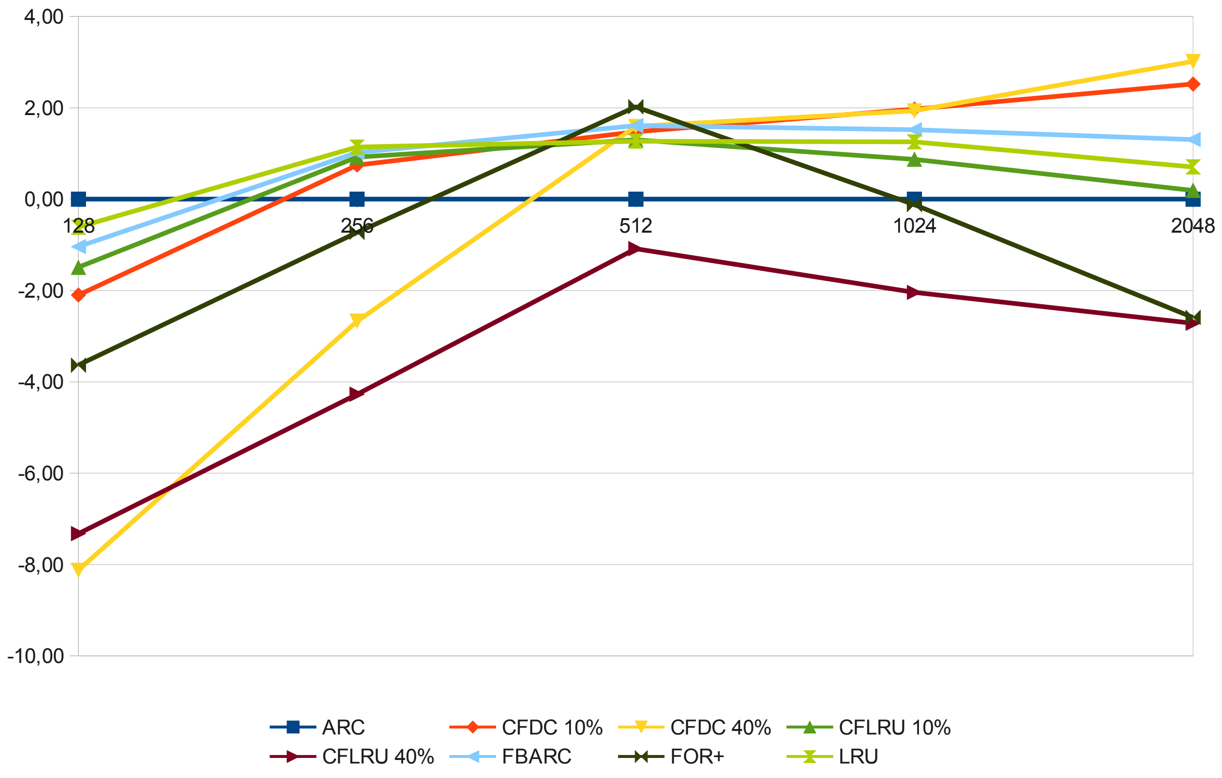


Figure 7.14: Relative change compared to ARC

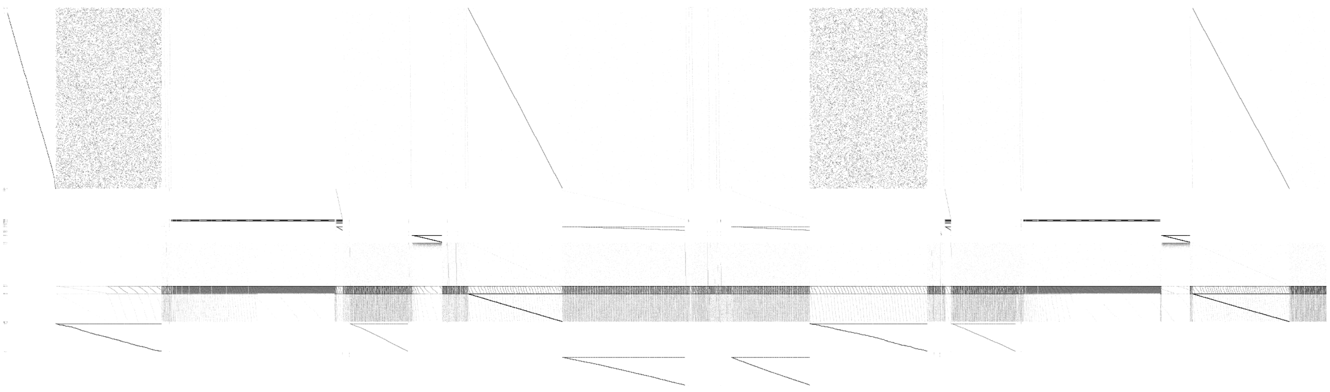


Figure 7.15: Trace H scatter diagram (some portions left out). TPC-H blocktrace - mostly read only access pattern. Complex analytical queries are executed that focus only on a few relations at a single time.

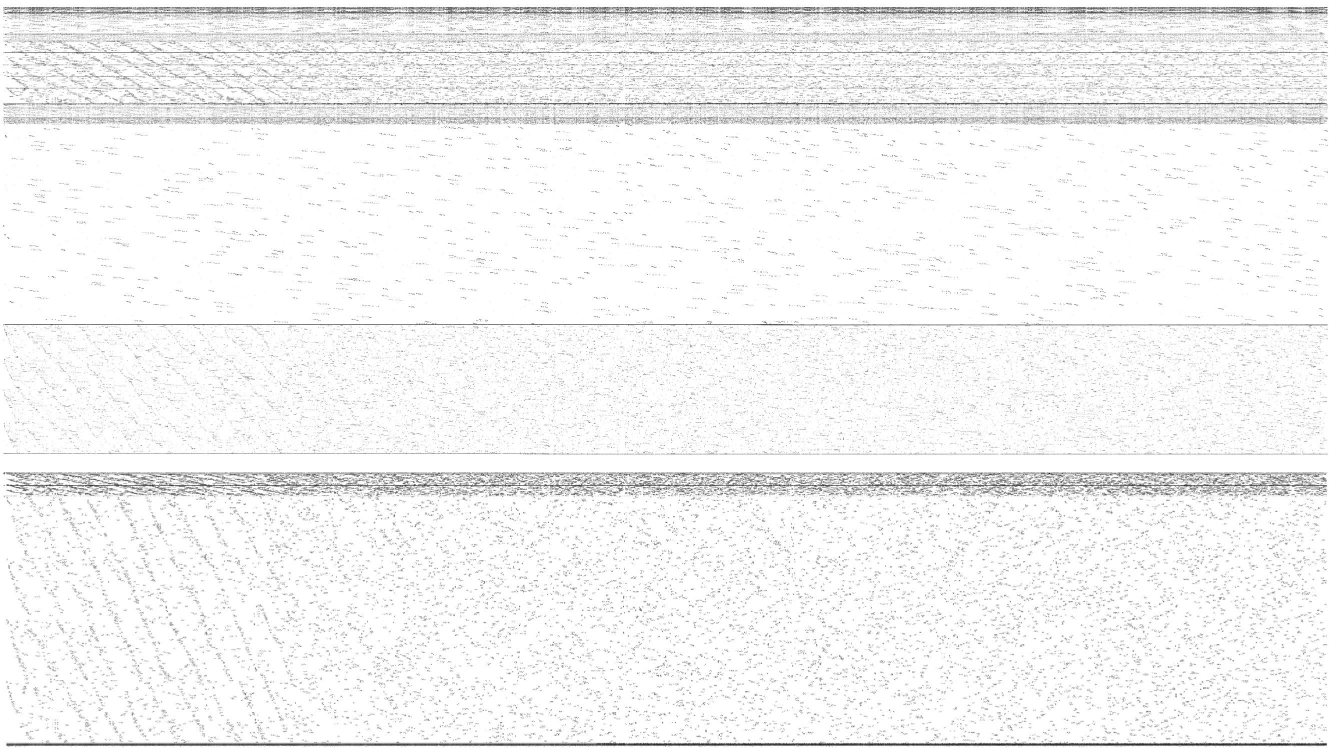


Figure 7.16: Trace C scatter diagram. The typical TPC-C blocktrace is depicted, showing the accesses of all relations (whole DB).

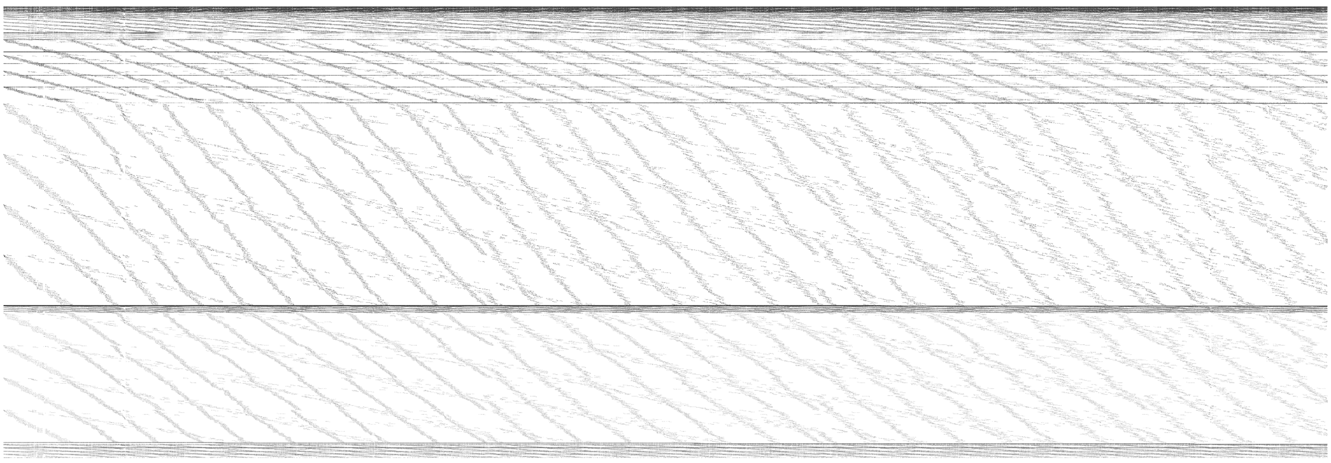


Figure 7.17: Trace C_d scatter diagram. The typical TPC-C blocktrace is depicted, showing only the accesses of the delivery relation.

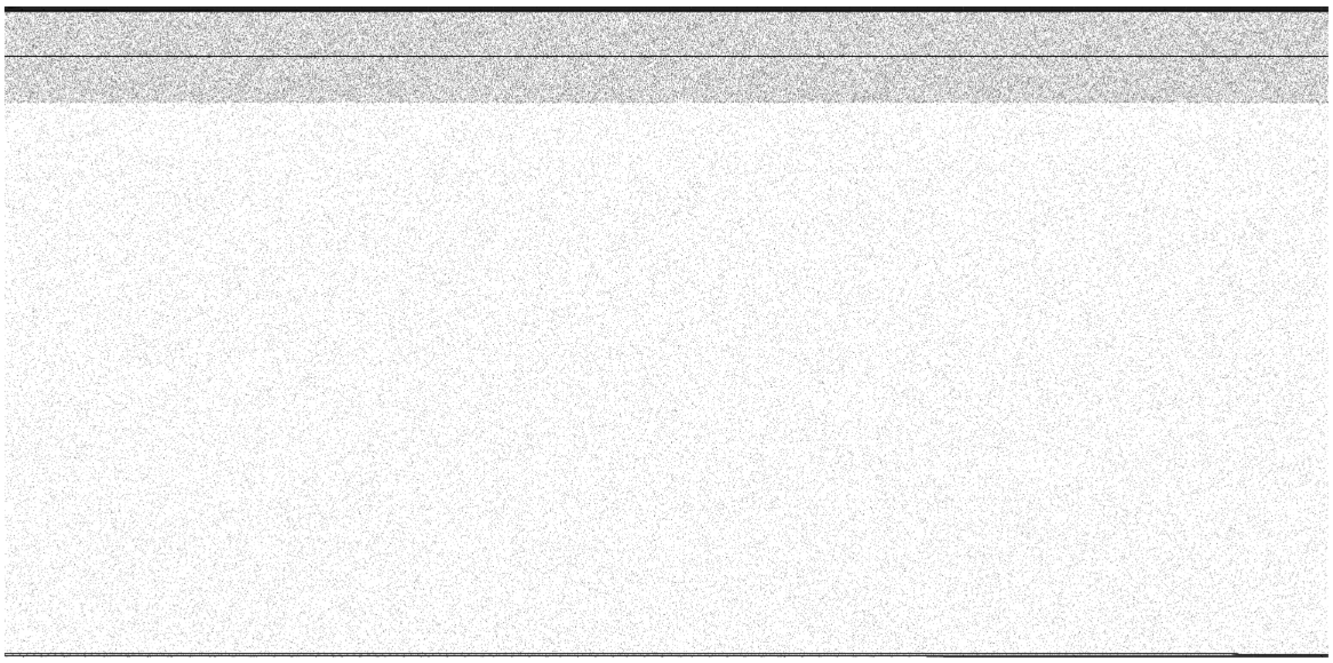


Figure 7.18: Trace B scatter diagram. Pgbench blocktrace showing the TPC-B access pattern. This pattern shows a uniform distribution of accesses.

computational-intensiveness of all clustering strategies. FBARC's lightweight clustering algorithm is based on the observation that on Flash devices random write operations that are restricted on a small address region are executed with close to sequential performance. This trend is also valid with the newest generation of devices and will persist. In addition, ARC and FBARC are the only scan-resistant buffer management strategies among those under test. Such property is especially relevant for Data Warehousing systems where large sequential scans may be followed by large sequential update streams. Moreover sequential patterns are even more common for new storage technologies since new algorithms target write sequentialization.

8 Related Work

Contents

8.1 Multi Version Databases **155**
8.1.1 Snapshot Isolation 155
8.1.2 Indexing in MV-DBMS 156
8.2 Temporal Databases **156**
8.3 In-Memory Databases **156**
8.4 Log Based Storage Management **157**
8.5 Snapshot Isolation Append Storage **157**
8.6 Flash Storage **158**
8.7 Flash Translation Layer **158**
8.8 Buffer Management **159**

8.1 Multi Version Databases

As a disruptive technology asymmetric storage such as Flash SSDs became the established storage media of modern database management systems in the last decade. Snapshot Isolation (SI) and Multi Version Concurrency Control (MVCC) enjoy broad system support. Some commercial and open source systems using MVCC are: Ingres, Microsoft SQL Server, Oracle, Oracle Berkeley DB, PostgreSQL. Some of them implement SI as a separate isolation level, while others use it to facilitate serializable isolation.

Concepts of concurrency control mechanisms in database systems are described in [6].

The theory of MVCC is presented by Bernstein and Goodman in [5]. There exist different approaches to MVCC [105] such as *Timestamp Ordering*, *Multi-version Two-Phase Locking* and *Snapshot Isolation*.

8.1.1 Snapshot Isolation

As proposed in [4] by Bernstein et al Snapshot Isolation (SI) may produce histories that are not serializable. SI is implemented in PostgreSQL [93]. An overview of its implementation in PostgreSQL is given in [104]. Serializable SI (SSI) is proposed in [14], based on read/write dependency serialization graphs and graph testing algorithms. SSI is refined with precisely SSI (PSSI) in [94]. A performance comparison between different MVCC algorithms is provided in [16]. [76] and [59] offer insights to the implementation details of SI in Oracle and PostgreSQL.

The scheme of organizing data item versions in simple chronologically ordered chains has been proposed in [17] and explored in [91, 11] in combination with MVCC algorithms and special locking approaches.

The MySQL InnoDB multi-versioning scheme stores older versions of data items in *rollback segments* [79]. On a request of an older version it is recreated using undo records.

The Microsoft SQL Server implements SI for the isolation levels *snapshot* and *Read committed snapshot* based on row versioning [105]. Update and delete operations create older versions of the affected rows and store them in a temporal database. Once no running transaction is required to read the older versions, they are garbage collected.

8.1.2 Indexing in MV-DBMS

The application of version oblivious indexing on a multi-versioned dataset is per se trivial, yet leveraging discrete properties of the multi versioned dataset is a challenging task. An overview of approaches on indexing in MV-DBMSs is provided in [55]. One of the earlies approaches for indexing multi-version datasets are Time-Split B+Trees (TSB-tree) [71, 72]. A modified version of $B^{+} - tree$ is proposed in [115]: index nodes yielding a validity period and data nodes with different record formats are introduced. Aiming to provide a general index structure for MV-DBMS the Multi-Version B-Tree is proposed in a series of papers by Haapasalo et al. [45, 42, 43, 44].

InnoDB distinguishes between primary and secondary indexes. Primary indexes are updated in-place and secondary indexes are updated by creating new index entries and marking the old entry invalid [79].

8.2 Temporal Databases

Temporal Databases (TMP-DBs) manage historical data. [106] describes approaches to temporal databases. Historical data is characterized by data that can be mapped to a discrete moment or timespan. The concept of time travel DBMSs that enable the user to execute historical queries is discussed in [110].

Temporal databases differ significantly from real-time databases, where transactions have deadlines or timing constraints, described in a survey in [85].

In comparison to a TMP-DB the MV-DBMS contains versions of data items that represent the *current state* of the DB. Versions are utilized to determine the correct/current state of the DB given for a transaction (transactional time). Each data item is represented by a non-empty set of tuple versions. Each running transaction is allowed to access at most one tuple version which represents the most recent state of the data item for that transaction.

In TMP-DBs updates to data items are rare and insert operations dominate. TMP-DBs handle data involving the dimension of time, including a temporal data model and an adapted version SQL [107]. An example for a TMP-DB is a warehouse that processes OLAP workload (business intelligence). In the case of TMP-DBs, the user is aware of the historical data set, which is not the case in MV-DBMSs.

8.3 In-Memory Databases

Recently *In-Memory Database Management Systems* (IM-DBMS) emerge as a forthcoming trend [31, 60, 1, 7, 112]. IM-DBMS operate on memory resident data and keep the database in-memory at any given time of the operation. Persistent storage media is used for logging and durability of the data.

Such approaches employ variants of versioning and version organization which are orthogonal to the work presented here.

A recent trend is the combination of OLTP and OLAP workload using in-memory MV-DBMS that create a *real-time business intelligence*.

SAP Hana [31] alleviates the need for a separation of the business intelligence (OLAP) and the operational system (OLTP) in two separate (server-) systems. The database is held in-memory and persistent storage is used for logging.

Hyper [60] separates OLAP from OLTP workload by the creation of virtual memory *snapshots* utilizing copy on write methods.

Although main memory prices have dropped (factor of 10 every 5 years [1]) they are still significantly higher than those of Flash storage media. In addition the energy consumption of DRAM is higher than that of Flash storage. For recovery and logging those systems also require additional persistent storage media.

Hekaton [1] describes a different approach to serializability based on visibility range definitions. Another approach to IM-DBMSs is proposed in [112].

8.4 Log Based Storage Management

Log-structured file systems in Linux are presented in [64] and [95]. FlexFS [65] is a file system that is designed for MLC NAND Flash. File systems are designed to be applicable for general software and do not address special properties of DBMSs.

[91, 11, 17] explore database related approaches to LbSMs. The authors also investigate approaches to garbage collection. Other non-in-place update strategies for Flash-based transaction systems are presented in [116]. LbSM approaches using delta stores still require relatively expensive merge operations and generate overhead on read accesses [80]. The applicability of LbSM approaches for asymmetric storage technologies such as Flash devices has been partially addressed in [109, 7, 108]. Hyder [7] is a transactional record manager for Flash storage. The approach in [61] stores updates in a delta store that is later merged with the main data store.

A common feature of these approaches is that they use page-granularity.

SIAS employs tuple-granularity much like the approach proposed in [11]. The latter 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 is achieved. Alternatively, given tuple-granularity, multiple new tuple-versions can be packed on a new page and written together once. In the area of operating systems log storage approaches at file system level for hard disk drives have been proposed in [96].

Approaches to group commit as described in [48] integrate the database and the log structure. Multiple transactions are delayed and committed with a single disk I/O. This work separates the database as storage and the log as the recovery control mechanism and does not delay transactions.

8.5 Snapshot Isolation Append Storage

In [34] a tuple version management scheme, called SI-CV, is adopted that co-locates tuple versions created by individual transactions. This aims at the reduction of random reads on Flash. Although this introduces several improvements, a main issue is not addressed by the approach: updated tuple versions are still invalidated in-place, which causes physical in-place updates.

SIAS-Vectors (Section 4.4) algorithmically integrates LbSM in tuple granularity into the MV-DBMS. It is demonstrated in [33] (video available online [103]). SIAS-Vectors no longer follows the paradigm of addressing each tuple version individually: all versions of a tuple (data item) receive the same unique identifier, the so called virtual ID (VID), which identifies the data item as a whole. SIAS-Chains introduces a new paradigm to version management, improves on the version invalidation scheme and adapts the indexing scheme. The existence of a valid successor of a tuple version implicitly invalidates the previous tuple version. In-place updates are conceptually avoided. SIAS-Chains co-locates recently inserted tuple versions. SI-CV [34] creates a different kind co-location of tuple versions. Tuple versions that are created by individual transactions are placed on individual buffer pages and each transaction receives a reserved buffer page, whereas SIAS-Chains (and SIAS-Vectors) co-locates tuple versions of different transactions. It is not required to reserve a buffer page for each individual transaction.

8.6 Flash Storage

An overview of Flash storage properties is given in [18], design and performance trade-offs are discussed in [3]. In [90] we give an overview of MV-DBMSs on modern storage. We provide an overview of append storage (LbSM) in MV-DBMSs on Flash in [36], which we extend in [35], [37]. In [38] we present read optimizations for LbSM in multi-version databases on Flash.

In [91] the configuration of RAID systems based on Flash Drives is evaluated, aiming at building up large and scalable storage out of relatively small but fast SSDs. The I/O parallelism of Flash drives can't be leveraged by modern RAID controllers. Software configurations that utilize the processing power of modern CPUs achieve greater performance (throughput, latency) than server class RAID controllers. This was explained in more detail in Section 2.5.

Rules of thumb in data engineering describing optimizations for legacy storage, such as the trend towards increasing block sizes, are provided in [40]. A recipe for building Flash-based data intensive systems is given in [47].

8.7 Flash Translation Layer

The FTL's main task is to provide a mapping of logical and physical addresses. Approaches to the mapping scheme can be classified as Page-level mapping schemes, Block-level mapping schemes or Hybrid-/Log-Block-mapping schemes. An evaluation and comparison of different FTLs is provided in [20] and [75]. In the past numerous designs of FTLs have been proposed. For instance [86], [41], [67], [66], [69], [74] etc..

DFTL is a page-mapping FTL and is introduced in [41]. There are multiple Flash simulation frameworks such as FlashSim [63] or DiskSim [27].

Ongoing research on omitting certain on-device FTL functionalities assigns the accessing layers more control over the Flash chips [8]. An approach that is not using the block I/O interface is presented in [84], [12] presents a hybrid approach which can bypass the on-device FTL.

Specialized Flash Server Storage such as FusionIO [2] moves the FTL from a device into the driver.

[46] demonstrates an on-line Flash simulator, delivering direct access to Flash chips. NoFTL completely removes the on-device FTL, enabling the application to take full control of the Flash storage device. The integration of the FTL functionality into the DBMS, providing native access to Flash storage yields significant performance benefits.

8.8 Buffer Management

Database buffer management is a traditionally well-studied field; a broad overview is provided in [30]. We distinguish between *general purpose* (FIFO, LRU [26], ARC [77]) and *special purpose* algorithms.

General purpose cache replacement algorithms can be used at any layer in the memory hierarchy. They, however, do not utilize special system knowledge to speed up accesses.

Special purpose algorithms on the other hand are designed to use system information and circumvent restrictions of the respective target area. They can be further sub-divided into:

- *Embedded algorithms* [62, 54] are designed to use low-level system information for devices with limited computational resources. An elevator algorithm for example needs to know where the disk head is positioned at the moment to decide which block to access next, or an FTL knows if a write will need a full merge or a switch merge.
- *Clustering algorithms* [83, 87, 53] target the spatial locality by grouping dirty in-buffer pages into clusters and making the eviction decision based on a composite per-cluster score.
- *Page State algorithms* [73, 57, 56, 87] make use of the page's state in the decision which page to evict and can adapt to different kinds of access. That way it can address the read/write asymmetry on Flash media. CFLRU [87], CFDC [83] and FOR/FOR+ [73] are cache replacement strategies that fall in the class of *page state* dependent replacement algorithms. The buffer strategies ARC, CFLRU, CFDC, FOR, DULO and BPLRU are described in detail in Chapter 7.
- *Persistent buffer extensions* [28, 15, 58] mainly provide long term archiving of data.

9 Summary and Future Work

Contents

9.1 Summary and Contributions **161**
9.2 Future Work **164**
9.2.1 Garbage Collection 164
9.2.2 Thinking Beyond the Block Interface 164
9.2.3 Compression of Sorted Runs 164

9.1 Summary and Contributions

Modern DBMSs fail to exploit the key properties of asymmetric Flash storage. Performance potential is lying idle and durability of the storage media is shortened which ultimately leads to higher costs. This thesis is a remedy for those shortcomings, making the DBMS aware of the underlying asymmetric Flash storage. Existing assumptions on design and architecture of modern MV-DBMSs are questioned and revised.

The *Snapshot Isolation Append Storage* (SIAS) approach subsumes changes on the algorithmic and architectural design of the MV-DBMS to optimize for asymmetric Flash storage. The combination of the MV-DBMS, multi-versioning concurrency control (MVCC) and append/log-based storage management (LbSM) on Flash storage delivers optimal performance figures which are needed to satisfy the urgent demand in scalable performance for modern DBMSs. We exploit the performance potential of the asymmetric Flash storage and increase its durability.

The contributions of this dissertation are:

- The *Snapshot Isolation Append Storage* (SIAS) approach subsumes changes on the algorithmic and architectural design of the MV-DBMS with the focus on asymmetric Flash storage. SIAS is a compilation of architectural design changes and algorithms that are designed for the specific properties of asymmetric storage devices such as SSDs. Beginning with the transaction processing and multi-version concurrency control, the transaction manager, the version organization and related access methods are redesigned. Simplified buffer management is incorporated into append storage management.

- SIAS changes the *invalidation model* to facilitate out-of-place invalidation (Section 4.1). SIAS introduces a new version organization scheme that does not rely on in-place invalidation.

- SIAS introduces a *new versioning scheme* that addresses tuple versions of a data item as a unique set (Section 4.3). Access methods such as scans and indexes are adapted and optimized.

- SIAS enables append storage management in *tuple granularity* (Chapter 6). Logical appends of tuple versions are enabled, coupled with the buffer manager physical appends are executed in page granularity.

- SIAS couples a simplified buffer management with append storage management by performing write retention (Chapter 7).
- The minimal invasive *Flash based adaptive replacement cache* (FBARC) has been developed (Chapter 7) which can be applied to in-place update DBMSs. With the FBARC buffer management, that employs static grid clustering the efficiency of semi-sequential writes on Flash storage is explored.

Existing assumptions on design and architecture of modern MV-DBMSs are questioned and revised:

- Clustering: clustering, as a read optimization, incurs in-place updates and creates mixed loads. Keeping a dataset clustered on Flash is not feasible. The in-place updates negate the positive effects of the clustering. Append storage is more suitable for asymmetric storage.
- Free Space Management: maintaining free space per page is not feasible for LbSM on asymmetric storage. It inflates space consumption and avoids efficient cache usage.
- The trend towards larger I/O transfer units (large page I/O) does not hold for Flash storage. Random reads are exceptionally fast on small transfer units.
- Version Organization Scheme: organization of tuple versions, addressing each of them as individual data items along with two-place invalidation, incurs in-place updates of visibility meta-data.
- In-Place Invalidation: the versioning model of in-place invalidation creates small in-place (random) updates which are unsuitable for the properties of Flash storage.
- In-Place Storage Management: single version schemes and in-place update approaches hinder concurrent access and parallelism.
- Logical Page Append Storage Management as an additional layer: LbSMs that operate in page granularity create write amplification (smallest granularity to append is a page, even if just a single bit in the page is updated).
- Buffer Managers Optimizing for Hitrate: optimizing for locality and grouping of writes, forming small sequential writes (or appends) becomes a major point, especially on in-place storage management.
- Read optimized datastructures: datastructures that optimize for HDD read access have to optimize for Flash storage write access.

The accumulation and integration of the presented techniques into SIAS lead to:

- Reduction of write amplification: SIAS leads to a significant reduction of the write amplification that is incurred by the version organization and the invalidation scheme.
- Less host writes: SIAS writes less data. Tuple versions are the net amount of data.
- Less space requirement: the amount of occupied space is reduced since less net amount of data has to be written and no free space reservation is applied.
- Avoidance of in-place updates caused by the version organization and invalidation scheme.

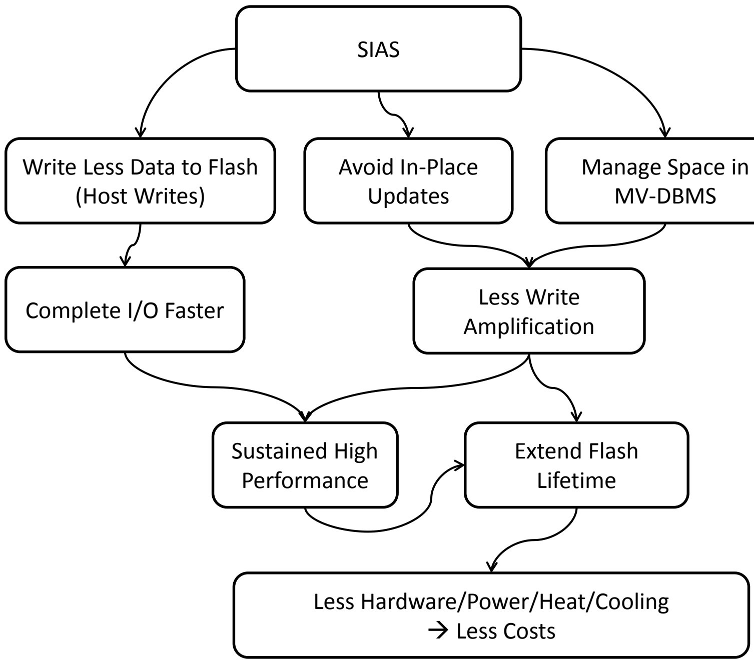


Figure 9.1: Benefits of SIAS in a nutshell.

- Complementing the FTL: more efficient garbage collection and wear leveling due to Flash-awareness.
- Performance benefits: Higher transactional througput with lower response times.
- Scalability: SIAS scales with more warehouses on TPC-C loads.
- Complete I/O faster: write less data and transfer more tuple versions (no free space reservation).
- Lower latency: I/O is completed faster, leading to latency reduction and increased transactional throughput on high loads
- Higher longevity: enhanced lifetime of Flash cells due to less host writes.
- Shifting the knee-point: lower latency and higher throughput allows a higher tolerance for peak workloads.
- The reported benefits of SIAS are summed up and depicted in Figure 9.1.

9.2 Future Work

9.2.1 Garbage Collection

Approaches to garbage collection in LbSMs are an interesting research topic. As described in Section 4.6 the steps performed during garbage collection involve:

- (i) finding the candidate (victim-) page that is chosen to be garbage collected,
- (ii) reinsertion of valid tuple versions and
- (iii) discarding of dead tuple versions of that page.

All three steps offer interesting points of research, providing research on advanced data placement. For example, hot and cold data separation and archiving.

This offers research questions such as: *"how to optimize for garbage collection efficiency?"*; and *"is it beneficial to place tuple versions that are updated at comparable rates on the same page?"*

In addition the synergy between garbage collection, LbSM free space management and the version organization has to be considered.

9.2.2 Thinking Beyond the Block Interface

The SIAS approach is working on tuple version granularity and assumes a block device interface.

Another point of research is it's combination with the NoFTL approach. It has been shown that removing the block device interface, thus moving more control over the Flash storage into the DBMS is beneficial for performance and durability.

Utilizing direct access to the Flash memory involves changing the LbSM component of SIAS and directs full control of the Flash memory to SIAS.

It has to be analyzed how possible effects of this combination influence the Flash memory (endurance, longevity) and the MV-DBMS (throughput, latency).

9.2.3 Compression of Sorted Runs

Leveraging the properties of sorted runs theoretically promises further improvements in terms of write performance and I/O behavior. Sorted runs define a finite and locally restricted dataset and tuples are sorted, hence tuple values correlate on tuple versions which are alike. Sorted runs can also be compressed when attributes only differ in small delta changes or contain common properties.

Integrating the compression into the MV-DBMS offers a possible reduction of header information, since a sorted run can also be seen as one large block of pages instead of multiple single pages that yield individual headers. The principle of using minimal headers for *very large tuples* is already applied in practice, e.g. as a *toast* relation in PostgreSQL [92]. Also each sorted run contains a finite set of tuple versions that differ in only a single attribute. Compression of sorted runs can reduce the amount of written data but increases the complexity of each read access. The idea of such a compression of a data-portion that is persisted on the Flash storage has been implemented most recently in hardware controllers of Flash SSDs [101].

Bibliography

[1] *Hekaton: SQL Server's Memory-Optimized OLTP Engine*. ACM International Conference on Management of Data, June 2013.

[2] Going beyond ssd: The fusionio software defined flash memory approach, 2013. www.fusionio.com/white-papers/beyond-ssd/, Last accessed february 8th, 2016.

[3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, pages 57–70, 2008.

[4] 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.

[5] P. A. Bernstein and N. Goodman. Multiversion concurrency controlâĂŤtheory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[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] M. Bjørling, P. Bonnet, L. Bouganim, and N. Dayan. The necessary death of the block device interface. Technical report, IT-Universitetet i København, 2012.

[9] blkparse I/O tool. http://linux.die.net/man/1/blkparse. Last accessed february 8th, 2016.

[10] blktrace I/O tool. http://www.cse.unsw.edu.au/ aaronc/iosched/doc/blktrace.html. Last accessed february 8th, 2016.

[11] 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.

[12] P. Bonnet, L. Bouganim, I. Koltsidas, and S. D. Viglas. System co-design and data management for flash devices. In *Proc. VLDB 2011*, 2011.

[13] L. Bouganim, B. T. Jónsson, and P. Bonnet. uflip: Understanding flash io patterns. *CoRR*, abs/0909.1780, 2009.

[14] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD*, SIGMOD '08, pages 729–738, New York, NY, USA, 2008. ACM.

[15] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang. SSD bufferpool extensions for database systems. *PVLDB*, 3(2):1435–1446, 2010.

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

[17] 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*, June 1982.

[18] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. SIGMETRICS '09, pages 181–192. ACM, 2009.

[19] S. Chen, P. B. Gibbons, and S. Nath. Rethinking database algorithms for phase change memory. In *Proc. CIDR*, pages 21–31, 2011.

[20] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. A survey of flash translation layer. *J. Syst. Archit.*, 55:332–343, May 2009.

[21] A. Croker and D. Maier. A dynamic tree-locking protocol. In *Data Engineering, 1986 IEEE Second International Conference on*, pages 49–56, Feb 1986.

[22] Database Test Suite DBT2. http://osdldbt.sourceforge.net. Last accessed february 8th, 2016.

[23] Database Test Suite DBT3. http://sourceforge.net/projects/osdldbt/files/dbt3/. Last accessed february 8th, 2016.

[24] Database Test Suite pgbench. http://www.postgresql.org/docs/devel/static/pgbench.html. Last accessed february 8th, 2016.

[25] Dell eMLC Specification. www.dell.com/downloads/global/ products/pvaul/en/solid-state-drive-faq-us.pdf, Last accessed february 8th, 2016.

[26] P. J. Denning. Working sets past and present. *IEEE Trans. Softw. Eng.*, 6:64–84, Jan. 1980.

[27] DiskSim Simulation Environment. http://www.pdl.cmu.edu/DiskSim/. Last accessed february 8th, 2016.

[28] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson. Turbocharging dbms buffer pool using ssds. In *Proc. SIGMOD*, pages 1113–1124, 2011.

[29] P. Dubs, I. Petrov, R. Gottstein, and A. Buchmann. Fbarc: I/o asymmetry aware buffer replacement strategy. In *Fourth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2013.

[30] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, Dec. 1984.

[31] 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.

[32] B. S. Gill and D. S. Modha. Wow: wise ordering for writes - combining spatial and temporal locality in non-volatile caches. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 10–10, 2005.

[33] 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*, pages 624–627. OpenProceedings.org, 2014.

[34] R. Gottstein, I. Petrov, and A. Buchmann. Si-cv: Snapshot isolation with co-located versions. In R. Nambiar and M. Poess, editors, *Topics in Performance Evaluation, Measurement and Characterization*, volume 7144 of *Lecture Notes in Computer Science*, pages 123–136. Springer Berlin / Heidelberg, 2012.

[35] R. Gottstein, I. Petrov, and A. Buchmann. Append storage in multi-version databases on flash. In *Big Data*, pages 62–76. Springer Berlin Heidelberg, 2013.

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

[37] R. Gottstein, I. Petrov, and A. Buchmann. Multi-version databases on flash: Append storage and access paths. *International Journal On Advances in Software*, 6(3 and 4):321–328, 2013.

[38] R. Gottstein, I. Petrov, and A. Buchmann. Read Optimisations for append storage on Flash. In *Proceedings of the 17th International Database Engineering & Applications Symposium*, pages 106–113. ACM, 2013.

[39] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. In *Proceedings of the 3rd international workshop on Data management on new hardware*, DaMoN '07, pages 6:1–6:9, New York, NY, USA, 2007. ACM.

[40] J. Gray and P. Shenoy. Rules of thumb in data engineering. In *Proc. ICDE'00*, pages 3–10, 2000.

[41] A. Gupta, Y. Kim, and B. Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proc. ASPLOS*, ASPLOS XIV, pages 229–240, 2009.

[42] T. Haapasalo, I. Jaluta, B. Seeger, S. Sippu, and E. Soisalon-Soininen. Transactions on the multiversion b+-tree. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 1064–1075, 2009.

[43] T. Haapasalo, S. Sippu, I. Jaluta, and E. Soisalon-Soininen. Concurrent updating transactions on versioned data. In *Proceedings of the 2009 International Database Engineering & Applications Symposium*, IDEAS '09, pages 77–87, 2009.

[44] T. K. Haapasalo. Accessing multiversion data in database transactions. In *Doctoral Dissertation, Aalto University*, TKK-CSE-A5/10, 2010.

[45] T. K. Haapasalo, I. M. Jaluta, S. S. Sippu, and E. O. Soisalon-Soininen. Concurrency control and recovery for multiversion database structures. In *Proceedings of the 2nd PhD workshop on Information and knowledge management*, PIKM '08, pages 73–80, 2008.

[46] 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.

[47] J. He, A. Jagatheesan, S. Gupta, J. Bennett, and A. Snavely. Dash: a recipe for a flash-based data intensive supercomputer. In *SC'10*, pages 1–11, 2010.

[48] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter. Group commit timers and high volume transaction systems. In *Proceedings of the 2Nd International Workshop on High Performance Transaction Systems*, pages 301–329, London, UK, UK, 1989. Springer-Verlag.

[49] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 10. ACM, 2009.

[50] Intel X25-E 64GB Vendor Specification. http://ark.intel.com/products/56596/intel-ssd-x25-e-series-64gb-2_5in-sata-3gbs-50nm-slc. Last accessed february 8th, 2016.

[51] IOWatcher. http://masoncoding.com/iowatcher/. Last accessed february 8th, 2016.

[52] S. Jiang. Dulo: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *In USENIX Conference on File and Storage Technologies (FAST*, 2005.

[53] P. Jin, Y. Ou, T. Härder, and Z. Li. Ad-lru: An efficient buffer replacement algorithm for flash-based databases. *Data Knowl. Eng.*, 72:83–102, Feb. 2012.

[54] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee. Fab: Flash-aware buffer management policy for portable media players. In *IEEE Transactions on Consumer Electronics, 52*, pages 485– 493. IEEE, 2006.

[55] K. Jouini and G. Jomier. Indexing multiversion databases. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 915–918. ACM, 2007.

[56] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha. Lru-wsr: integration of lru and writes sequence reordering for flash memory. volume 54, pages 215 – 1223, Aug. 2008.

[57] H. Jung, K. Yoon, H. Shim, S. Park, S. Kang, and J. Cha. Lirs-wsr: integration of lirs and writes sequence reordering for flash memory. In *ICCSA'07*, pages 224–237. Springer-Verlag, 2007.

[58] W.-H. Kang, S.-W. Lee, and B. Moon. Flash-based extended cache for higher throughput and faster recovery. *Proc. VLDB Endow.*, 5(11):1615–1626, 2012.

[59] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases*, pages 134–143. Morgan Kaufmann, 2000.

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

[61] A. Kemper and T. Neumann. Hyper: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In S. Abiteboul, K. Böhm, C. Koch, and K.-L. Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206. IEEE Computer Society, 2011.

[62] H. Kim and S. Ahn. Bplru: a buffer management scheme for improving random writes in flash storage. In *Proc. FAST'08*, pages 16:1–16:14, 2008.

[63] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. In *In Proc. SIMUL'09*, pages 125–131, 2009.

[64] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[65] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim. Flexfs: A flexible flash file system for mlc nand flash memory. In *USENIX Annual Technical Conference*, pages 1–14, 2009.

[66] S.-W. Lee, W.-K. Choi, and D.-J. Park. Fast: An efficient flash translation layer for flash memory. In X. e. a. Zhou, editor, *Emerging Directions in Embedded and Ubiquitous Computing*, LNCS. 2006.

[67] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM TECS*, 6(3), July 2007.

[68] Y. Li, J. Xu, B. Choi, and H. Hu. Stablebuffer: optimizing write performance for dbms applications on flash devices. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 339–348. ACM, 2010.

[69] S.-P. Lim, S.-W. Lee, and B. Moon. Faster ftl for enterprise-class flash memory ssds. In *SNAPI)*, 2010.

[70] Linux Software Raid - mdadm. http://www.linuxcommand.org/man_pages/mdadm8.html. Last accessed february 8th, 2016.

[71] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, SIGMOD '89, pages 315–324, New York, NY, USA, 1989. ACM.

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

[73] Y. Lv, B. Cui, B. He, and X. Chen. Operation-aware buffer management in flash-based systems. In *SIGMOD '11*, pages 13–24, 2011.

[74] D. Ma, J. Feng, and G. Li. Lazyftl: a page-level flash translation layer optimized for nand flash memory. In *Proc. SIGMOD '11*, 2011.

[75] D. Ma, J. Feng, and G. Li. A survey of address translation technologies for flash memories. *ACM Comput. Surv.*, 46(3):36:1–36:39, Jan. 2014.

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

[77] N. Megiddo and D. S. Modha. ARC: a Self-Tuning, low overhead replacement cache. In *Proc FAST*, pages 115–130, 2003.

[78] C. Mohan. *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes.* IBM Thomas J. Watson Research Division, 1989.

[79] MySQL InnoDB Multi Versioning. http://dev.mysql.com/doc/refman/5.7/ en/innodb-multi-versioning.html. Last accessed february 17th, 2016.

[80] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree, 1996.

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

[82] Y. Ou and T. Härder. Clean first or dirty first?: a cost-aware self-adaptive buffer replacement policy. In *IDEAS '10*, pages 7–14, 2010.

[83] Y. Ou, T. Härder, and P. Jin. Cfdc: a flash-aware buffer management algorithm for database systems. In *ADBIS'10*, pages 435–449, 2010.

[84] X. Ouyang, D. W. Nellans, R. Wipfel, and D. Flynn. Beyond block i/o: Rethinking traditional storage primitives. In *HPCA*, pages 301–311, 2011.

[85] G. Özsoyoğlu and R. T. Snodgrass. Temporal and real-time databases: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 7(4):513–532, 1995.

[86] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim. A reconfigurable ftl architecture for nand flash-based applications. *TECS*, 7(4):38:1–38:23, 2008.

[87] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee. Cflru: a replacement algorithm for flash memory. In *CASES '06*, pages 234–241, 2006.

[88] I. Petrov, G. Almeida, A. Buchmann, and U. Gräf. Building large storage based on flash disks. In *Proceeding of ADMS 2010, in conjunction with VLDB 2010*, 2010.

[89] I. Petrov, R. Gottstein, G. Almeida, T. Ivanov, and A. Buchmann. Using flash ssds as primary database storage. Invited Talk, SPEC Benchmark Workshop, October 2010.

[90] I. Petrov, R. Gottstein, and S. Hardock. Dbms on modern storage hardware. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1545–1548. IEEE, 2015.

[91] I. Petrov, R. Gottstein, T. Ivanov, D. Bausch, and A. Buchmann. Page size selection for OLTP databases on SSD RAID storage. *Journal of Information and Data Management*, 2(1):11, 2011.

[92] PostgreSQL - Toast Relation. http://www.postgresql.org/docs/9.5/static/storage-toast.html. Last accessed february 8th, 2016.

[93] PostgreSQL MV-DBMS. http://www.postgresql.org. Last accessed february 8th, 2016.

[94] S. Revilak, P. O'Neil, and E. O'Neil. Precisely serializable snapshot isolation (pssi). In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 482 –493, april 2011.

[95] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[96] 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.

[97] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10:26–52, February 1992.

[98] Samsung 3D-VNAND. http://www.samsung.com/global/business/semiconductor/minisite/ssd/downloads/document/samsung_ssd_850_pro_data_sheet_rev_1_0.pdf, Last accessed february 8th, 2016.

[99] Samsung 840 Pro Review by storagereview.com. http://www.storagereview.com/ samsung_ssd_840_pro_review. Last accessed february 16th, 2016.

[100] Samsung 840EVO SSD Vendor Specification. http://www.samsung.com/global/ business/semiconductor/ minisite/ssd/de/html/ssd840evo/specifications.html. Last accessed february 8th, 2016.

[101] Sandforce Flash Controller. http://www.seagate.com/de/de/products/ solid-state-flash-storage/flash-controllers/. Last accessed february 8th, 2016.

[102] Seekwatcher. https://oss.oracle.com/ mason/seekwatcher. Last accessed february 8th, 2016.

[103] SIAS-V Demo Video. http://dblab.reutlingen-university.de/tl_files/downloads/sias_v-inaction.mp4. Last accessed february 8th, 2016.

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

[105] A. Silberschatz, H. F. Korth, S. Sudarshan, et al. *Database system concepts*, volume 4. McGraw-Hill New York, 1997.

[106] R. T. Snodgrass. Temporal databases. In *IEEE computer*. Citeseer, 1986.

[107] R. T. Snodgrass. Developing time-oriented database applications in sql. 2000.

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

[109] R. Stoica, M. Athanassoulis, R. Johnson, and A. Ailamaki. Evaluating and repairing write performance on flash devices. In P. A. Boncz and K. A. Ross, editors, *Proceedings of the Fifth International Workshop on Data Management on New Hardware, DaMoN 2009, Providence, Rhode Island, USA, June 28, 2009*, pages 9–14. ACM, 2009.

[110] M. Stonebraker and G. Kemnitz. The postgres next generation database management system. *Communications of the ACM*, 34(10):78–92, 1991.

[111] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

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

[113] Transaction Processing Council Benchmark C. www.tpc.org/tpcc/spec/tpcc_current.pdf, Last accessed february 8th, 2016.

[114] S. C. Tweedie. Journaling the linux ext2fs filesystem. In *The Fourth Annual Linux Expo*, 1998.

[115] P. J. Varman and R. M. Verma. An efficient multiversion access structure. *IEEE Trans. on Knowl. and Data Eng.*, 9(3):391–409, May 1997.

[116] Y. Wang, K. Goda, and M. Kitsuregawa. Evaluating non-in-place update techniques for flash-based transaction processing systems. In *Database and Expert Systems Applications*, pages 777–791. Springer, 2009.