Universität Stuttgart

Multi-version Indexing for large Datasets with high-rate continuous Insertions

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von
Christian Riegger
aus Hechingen

Hauptberichter:	Prof. Dr.-Ing. Ilia Petrov
Mitberichter:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Tag der mündlichen Prüfung:	02. Juni 2023

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart

2023

Acknowledgments

Foremost, I would like to express my sincere gratitude to my supervisors Prof. Dr.-Ing. Ilia Petrov and Prof. Dr.-Ing. habil. Bernhard Mitschang. Ilia triggered my research interest in algorithm and data structures during my master's degree. He made the completion of this thesis possible by his support, expertise, encouragement and empathize for several personal misfortunes during the time. Bernhard Mitschang supported by inputs from a different perspective, what contributed to improvements and comprehensibility of this work.

I gratefully acknowledge the cooperation and scientific exchange with my colleagues in the Data Management Lab of Reutlingen University, Tobias Vinçon, Arthur Bernhardt and Christian Knödler, which yielded interesting research, novel approaches and collaborative publications.

I would like to thank my family and close friends for bearing with me all this time.

This research was partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program 'Services Computing'.

Contents

List of Abbreviations **9**

Zusammenfassung **13**

Abstract **15**

1. Introduction **17**

1.1. Motivation 18
1.2. Problem Statement 20
1.3. Contributions 22
1.4. Scientific Publications 24
1.5. Structure of this Thesis 26

2. Technical Background **27**

2.1. Characteristics of modern Hardware Technologies 28
2.1.1. Highly parallelized and decentralized processing units 28
2.1.2. Primary and Secondary Storage Characteristics 29
2.1.3. Testbed Hardware Metrics and Operating System . . . 34
2.2. Modern Workload Properties 34
2.2.1. Application of standardized Database Benchmarks . . . 35

2.3. Fundamental Design Decisions in Multi-Version Database Management Systems 37
2.3.1. Opportunities and Version Management in Multi-Version Concurrency Control 38
2.3.2. Estimated Base Table Cost Model 46
2.4. Additional Access Paths in Multi-Version DBMS 47
2.4.1. Version-oblivious Index Equality and Range Search . . 49
2.5. Improving Index Search Operation in Multi-Version DBMS . . 51

3. Related Work 53
3.1. Storage and auxiliary Index Management Structures 54
3.1.1. Ubiquitous B$^+$-Tree 54
3.1.2. Structures with Multi-Version Capabilities 60
3.1.3. Structures leveraging Modern Secondary Storage Hardware Characteristics 62
3.1.4. Applied Basic Structure: Partitioned BTree 63
3.1.5. Prototypical Environments 65
3.2. Summary and Conclusion 68

4. Multi-Version Partitioned BTree 69
4.1. Multi-Version Indexing Design Concepts 70
4.2. Write-Optimization with Partitioned BTrees 72
4.2.1. Data Structure and Components 73
4.2.2. Partition Switch & Sequential Write 86
4.2.3. Basic Operations 94
4.2.4. Cost Model 104
4.2.5. Summary 110
4.3. Version-Awareness with Multi-Version Partitioned BTree . . . 110
4.3.1. Applying Multi-Version Record Types 112
4.3.2. Version-Aware Record Sizes and Compression Techniques 114
4.3.3. Logical Version Chain in evolving MV-PBT 115
4.3.4. In-Partition Version Record Ordering Convention 119
4.3.5. Version-Aware Basic Operations 122

4.3.6. Index-Only Visibility Check 127
4.3.7. Summary 134
4.4. Workload Adaptiveness and Optimizations 134
4.4.1. Adaptive Data Reorganization and Indexing 135
4.4.2. Garbage Collection 142
4.4.3. Application of Auxiliary Filter Structures and Data Skipping 152
4.4.4. Summary 157
4.5. Conclusion 158

5. Arbitrary Approximate Membership Testing **159**
5.1. Problem Statement 160
5.2. Point Filter Techniques 161
5.3. Broadened Range Filter Techniques 163
5.3.1. Related Work 165
5.4. bloomRF: Unified Point-Range Filter for Arbitrary Approximate Membership Testing 167
5.4.1. Implicit Dyadic Trace Trees (DTT) 169
5.4.2. Prefix Hashing and Piecewise-monotone Hash Functions173
5.4.3. Error Correction Policies 177
5.4.4. Heuristical Tuning Advisor 181
5.4.5. bloomRF's False Positive Rate (FPR) Model 183
5.4.6. Basic Operations 184
5.4.7. Multi Attribute and Data Type Support 190
5.5. Experimental Evaluation of PRF 194
5.6. Conclusion 203

6. Application and System Integration **205**
6.1. Indexing in Relational Database Management Systems 206
6.1.1. Implementation Details of MV-PBT in PostgreSQL with SIAS 206
6.1.2. Experimental Setup 206
6.1.3. Evaluation and Selection of Baseline 207

6.1.4. Write-optimized Indexing in OLTP 211
6.1.5. Index-Only Visibility Checks dominate in HTAP 214
6.1.6. Summary 217
6.2. Storage Manager in Key/Value-Stores 217
6.2.1. Implementation Details of MV-PBT in WT 218
6.2.2. Implications on Secondary Indexing MV-PBT Stores . . 219
6.2.3. Experimental Setup 220
6.2.4. Experimental Evaluation 220
6.2.5. Summary 228
6.3. Conclusion 229

7. Summary and Research Opportunities 231
7.1. Summary of Contributions 232
7.2. Further Application Scenarios of MV-PBT 233
7.2.1. Time-Travel Query Processing 233
7.2.2. Blockchain Storage Structure in Enterprise Systems . . 234
7.2.3. Accelerators and Complex Memory Hierarchies 235

Bibliography 237

A. Appendix 259
A.1. Complex Memory Hierarchies 259
A.2. Technical Background: Flash in SSD 260
A.3. Example calculation of single B^+-Tree Cache Efficiency 262

List of Abbreviations

Abbreviation	Description
B^+-Tree	Balancing Tree Structure with Records in Leaves [BM70; Com79] (compare Section 3.1.1)
BF	Bloom Filter [Blo70] (compare Section 5.2)
bloomRF	Bloom Range Filter [MRBP23; RBMP20]
BPK	Bits per Key
Bw-Tree	Latch-Free Storage Management Structure based on B-Tree [LLS13; WPL+18]
CH-Benchmark	HTAP Benchmark based on TPC-C and TPC-H [CFG+11]
CP	Cached Partition
CPU	Central Processing Unit
DBMS	(Relational) (Multi-Version) Database Management System

Continued on next page

Table 1: Common abbreviations used in this thesis.

Continued from previous page

Abbreviation	Description
DI	Dyadic Interval
DTT	Dyadic Trace Tree
ETL	Extraction, Transformation and Load Process
FPR	False Positive Rate
GC	Garbage Collection
HDD	Hard Disk Drive
HOT	Heap-Only Tuple (Standard Base Table Organization in PostgreSQL [Pos21])
HTAP	Hybrid Transactional and Analytical Processing
I/O	Input- / Output-Operations (in the context of read or write to secondary storage)
IOPS	Input- / Output-Operations (I/O) per Second
K/V-Store	Key-/Value-Store
LBA	Logical Block Address
LSM	Log-Structured Merge Tree [OCGO96; SR12] (compare Section 3.1.3)
MV-PBT	Multi-Version Partitioned BTree [RP22; RVGP20]
MVCC	Multi-Version Concurrency Control
NO-tpmC	New-Order Transactions per Minute of TPC-C Workload [TPC10]
NVM	Non-Volatile Memory
OLAP	Online Analytical Processing

Continued on next page

Table 1: Common abbreviations used in this thesis. (Continued)

Continued from previous page

Abbreviation	Description
OLTP	Online Transaction Processing
Ops/s	Operations per Second
PBA	Physical Block Address
PBF	Prefix Bloom Filter
PBT	Partitioned BTree [RVP17b; RVP19] (based on [Gra03])
PMHF	Piecewise-Monotone Hash Function
Pnr	Partition Number
PRF	Point-Range Filter
$R_{I/O}$	Read I/O Costs
RA	Read Amplification
RAM	Random Access Memory (Primary Storage / Main Memory)
Rosetta	Robust Space-Time Optimized Range Filter for Key-Value Stores [LCK+20]
RQ	Research Question
SA	Space Amplification
SI	Snapshot Isolation
SIAS	Snapshot Isolation Append Storage [Got16] (compare Section 3.1.5.1)
SSD	Solid State Drive
SuRF	Succinct Range Filter [ZLL+18]

Continued on next page

Table 1: Common abbreviations used in this thesis. (Continued)

Continued from previous page

Abbreviation	Description
TPC	Transaction Processing Performance Council [TPC22]
TPC-C	Transaction Processing Performance Council Benchmark C [TPC10]
TPC-H	Transaction Processing Performance Council Benchmark H [TPC21]
tp{m,s}	Transactions per Minute, Second
TS	Timestamp
TT	Trace Tree
TX_{TS}	MVCC Transaction with Timestamp TS
VID	Virtual Identifier
$W_{I/O}$	Write I/O Costs
WA	Write Amplification
YCSB	Yahoo! Cloud Serving Benchmark [CST+10; RKD21]

Table 1: Common abbreviations used in this thesis.

Zusammenfassung

Die Eigenschaften moderner Arbeitslasten gegenüber Datenbanken orientieren sich an den Anforderungen von Geschäftsanwendungen, die auf Wissensgewinne und einen technologischen Vorsprung ausgerichtet sind. Diese Arbeitslasten zeichnen sich durch ein exponentielles Datenwachstum, einen hohen Anteil an kontinuierlichen Einfügungen und analytische Verarbeitung aus. Flash Sekundärspeicher sind eine wirtschaftliche Möglichkeit, mit großen Mengen modifizierbarer Daten umzugehen, wenn ihre Eigenschaften effizient genutzt werden. In diesem Kontext führt die Verwaltung von physisch materialisierten Tupel-Versionen in Basistabellen durch vorteilhafte Zugriffsmuster auf Sekundärspeichermedien zu einem kostengünstigen und skalierbaren transaktionalen Durchsatz.

Vorteilhafte Eigenschaften gelten jedoch kaum für gängige unversionierte sekundäre Zugriffspfade. Ihr tatsächlicher Nutzen wird durch Wartung sowie zusätzliche Kosten für Suche und Sichtbarkeitsprüfung begrenzt.

Mittels empirischer Methoden, also Literaturstudien und kontrollierter Experimente, werden spezifische Eigenschaften von modernen Arbeitslasten, Flash-basierten Speichertechnologien und modernster Datenverwaltungstechniken in Wissenschaft und Industrie gesammelt, um existierende Probleme, Forschungsmöglichkeiten und Herausforderungen in deren Interaktion zu identifizieren. Auf Basis der daraus abgeleiteten Erkenntnisse werden sowohl neue als auch ausgereifte Techniken betrachtet, mit dem

Ziel eine neue multi-versionierte Speicher- und Indexverwaltungsstruktur für die Eigenschaften moderner Speichertechnologien zu entwickeln. Durch standardisierten Arbeitslasten wird eine prototypische Implementierung in bekannten Systemen experimentell evaluiert.

Diese Arbeit leistet einen wichtigen Beitrag zur modernen Datenverwaltung mit Multi-Version Partitioned BTrees (MV-PBT). Ferner beinhaltet dies eine anfüge-basierte Tupel-Versions-Verwaltung für Arbeitslasten mit gleichbleibend hohen Dateneinspeisungsraten und analytischer Verarbeitung einer gemeinsamen Datenbasis, welche sich hauptsächlich auf einem Sekundärspeicher befindet. Erstens, der Ansatz verbessert die Selektivität sekundärer Zugriffspfade durch die Einführung von internen Sichtbarkeitsprüfungen. Dies führt bei gemischten Arbeitslasten zu einer Verdopplung des analytischen und zu einer Erhöhung von 14% des transaktionalen Durchsatzes. Zweitens, das Tupel-Aktualisierungsverfahren verbessert die Verwaltungskosten von Indizes durch das Hinzufügen logisch verketteter Versionen, womit ein um 47% erhöhter Durchsatz erzielt wird. Drittens, MV-PBT eignet sich aufgrund seiner Schreibeigenschaften und kosteneffizienten Suchvorgängen als Speicherverwaltungsstruktur, welche den Durchsatz im Vergleich zu den verbreiteten LSM-Bäumen verdoppelt. Viertens, die logische Verknüpfung von Versionsdaten erleichtert unabhängige Partitionierungs-, Reorganisations- und Wartungstechniken, was einen gleichbleibenden und um ein Vielfaches erhöhten Durchsatz für verschiedene Arbeitslasten ermöglicht. Letztens, bloomRF ermöglicht als neuartige, kostengünstige Punkt- und Intervall-Filtertechnik gleichbleibende Leistung für Such- und Wartungsoperationen in MV-PBT.

In dieser Arbeit werden die Vorteile einer multi-versionierten Hochleistungsindexverwaltung für die Eigenschaften moderner Arbeitslasten erläutert. Darüber hinaus kombinieren erweiterbare Designkonzepte auf Basis des B^+-Baums moderne Hauptspeichertechniken mit enormen Datenmengen und bilden die Grundlage für die Einbindung dezentraler Verarbeitungs- und Speichertechnologien. MV-PBT ist für ein breites Anwendungsspektrum geeignet und bildet einen ganzheitlichen Ersatz für bestehende Speicher- und Indexverwaltungsstrukturen.

Abstract

Trends in modern database workload properties are guided by business application needs with the characteristics of exponential growth of data, high-rate continuous insertions and analytical processing, aiming for knowledge gains and leading edges over competitors. Cheap Flash-based secondary storage devices provide an economic way to deal with massive amounts of modifiable data whenever their characteristics are efficiently leveraged. Thereby, it turned out that maintenance of physical materialized tuple version records in base tables not only scale with the number of concurrent transactions, but also provides beneficial access patterns to secondary storage devices, whereby asset and operational costs become manageable.

However, beneficial characteristics are hardly valid for common version-oblivious additional access paths. Their actual profit is limited by excessive maintenance as well as additional search and visibility check costs.

By means of empirical methods, i.e. literature studies and controlled experiments, characteristics of modern workloads, Flash-based storage hardware as well as state-of-the-art data management techniques in academia and industry are gathered in order to identify existing problems, research opportunities and challenges in mutual interactions. Based on derived findings, novel as well as matured techniques are considered to design a new kind of version-aware and hardware leveraging storage and index management

structure, which is prototypically implemented and integrated in well-known systems and experimentally evaluated by system performance benchmarks.

This thesis gives significant contributions in modern data management with Multi-Version Partitioned BTrees (MV-PBT) – i.e. append-based multi-versioned tuple maintenance, high-rate continuous insertion workloads with analytical processing on a common dataset instance, which comprises massive amounts of data on secondary storage devices. First, the approach massively improves selectivity of additional access paths by introduced index-only visibility checks, yielding 2× increased analytical and 14% transactional throughput in mixed workloads. Second, strict append-based and out-of-place replacement update schemes facilitate improved benefit by maintained indexes, by 47% improved throughput. Third, due to its near-optimal write characteristics and cost-efficient searches, the applied approach is highly qualified as storage management structure, with 2× increased throughput compared to widely used LSM-Tree. Fourth, logical linkage of version records facilitate independent partition, reorganization and maintenance techniques as well as robust performance characteristics for various workload properties, scaling up to orders of magnitude. Last, bloomRF, as a novel low-cost point-range filter technique, enables robust performance characteristics for search and maintenance operations in MV-PBT.

Contributions of this work unambiguously elaborate benefits of high-performant version-aware index management for recent developments in modern workload properties. Moreover, extendable design concepts on base of the ubiquitous B^+-Tree combine modern in-memory techniques, massive amounts of data and form a basis for recent trends in decentralized processing and storage hardware technologies. MV-PBT exhibits a broad range of applicability and facilitates a full substitution of matured storage and index management structures.

Chapter 1

Introduction

This introductory chapter firstly motivates the scope of action and significance to decision-makers in a general business context. Within this context, I share my personal experience, when I was involved as a trainee in the launch of a customer relationship management system and business warehousing solution upon an existing enterprise resource planning application. Upon that, consequent challenges of the motivational context are outlined and treated contributions are specified. A brief list of authors publications within this research context and following structure of this thesis close the chapter.

1.1. Motivation

Recent trends in business process requirements to information management systems result in new types of workloads on their most performance critical backbone – the database management systems (DBMS). Traditionally, DBMS workloads can be separated in different categories, such as online transactional processing (OLTP) or online analytical processing (OLAP). Characteristics of access patterns as well as effects on data management technologies massively differ. Emerging popularity of large scale, data intensive, real time analytical applications combine these characteristics in hybrid transactional and analytical processing workloads (HTAP) on the same dataset instance [HG20; MBL17; ÖTT17; RVGP20; SDA21].

An HTAP scenario brings several benefits to an enterprise in contrast to traditional data warehousing approaches, whereby data is **e**xtracted from the transaction processing system, **t**ransformed in a special query optimized form and **l**oaded to an analytical processing system (**ETL**-process).

First, real time data in HTAP scenarios improve and simplify quality as well as output of decision-making process in business operations [HG20; ÖTT17]. 'Greater situation awareness' and 'improved business agility' are required skills to stay competitive [Gar15].

> *As a personal experience, any delay in relevance of reported data reduces user experience and increases frustration. Furthermore, analytical outputs influence decisions in operative tasks, like forecasting, pricing or production planning and controlling. Afterwards or incorrectly maintained data in an OLTP-system deficiently affect business operations between ETL-processes or require enormous manual adjustments and manpower. Anyways, significant costs are affiliated to a company.*

Second, if a system is able to adopt the new HTAP-approach, an information management infrastructure of organizations could be simplified [Gar15; MBL17]. Less complex infrastructures are cheaper to maintain and evolve.

Last, global players have (almost) no operative down times. The extraction process probably stresses an OLTP-system and significantly shrinks transactional throughput for a long period of time[1].

Alongside HTAP-characterizations, upcoming technologies, markets, business cases and cloud services yield data-intensive and high-rate continuous insertion workloads. Internet of Things (IoT) platforms [Bos21; IBM21; Mic21a; Ora21b; SAP21; Ser21] handle data of billions of IoT-devices. The intention is by no means just storing massive amounts of data, rather a gain of knowledge and a technological distinct competitive edge. For instance, an automotive manufacturer could necessitate service intervals based on evaluated wear and sensor data correlation gathered by cloud IoT platforms.

Quality of service aspects in such cloud approaches are contractually fixed in service level agreements [Koh18]. Violation of these metrics, e.g. in transactional throughput, response times, availability or consistency, are associated with noticeably monetary costs [SOSM12]. A key driver for compliance in data-intensive tasks are naturally located in DBMS and their appropriated hardware resources. In this context, predominant approaches are evaluated in [HSB15]. Scaling-up means upgrading powerful components in single nodes, whereas scaling-out allows acquisition of more processing nodes.

Both approaches, especially in combination, enable management of a massive amount of data. Accompanying challenges in scale-out methods, like partitioning, communication, data locality and skew handling [CL16; RMU+14; Röd16; XY17; ZBS15], and scaling-up hardware [FZZ+19; KHL18; PDZ+18], are in scope of recent proposals. Several solutions focus on main-memory optimizations [DFI+13; FML+12; KN11; Pav14], e.g. in the area of key sorted indexing [KFM+15; LKN13; LLS13; ZCWJ21]. Essential prerequisites are large volatile main-memory capacities and powerful processing units as well as persistent non-volatile secondary storage resources. Increased main-memory (scale-up) or scaling-out the problem space by new data nodes for growing dataset sizes are costly in hardware, operation and administration [LHKN18].

[1] [Sup21] mentions a general job execution time of 2-4 hours.

Focusing only on main-memory leads to natural limits – in a technical as well as in an economical context [Lom18; NF20]. The much cheaper secondary storage technologies miss scaling principles in main-memory-oriented configurations and are limited to backup tasks and archiving. Facebook is a pioneer in introduction and evaluation of new solid state storage devices (SSD) and upcoming non-volatile memory (NVM) in massive data production setups [EGA+18; MWKM15]. Leveraging characteristics of new storage hardware in a complex memory hierarchy allow comparable performance to expensive up-scaled main-memory solutions and cheap – typically low main-memory equipped – data nodes. Establishing a technological edge by leveraging hardware characteristics beyond, could significantly reduce asset, administrative and operative costs.

In combination of these contexts, current storage manager and structures of (secondary) access paths encounter problems, outlined in the following section.

1.2. Problem Statement

Viewed in isolation, aspects of *application generated workloads* (Section 2.2), *characteristics of modern hardware technologies* (Section 2.1) and *fundamentals of DBMS designs* (Section 2.3) are well studied and partially state-of-the-art. Combining these aspects imply new challenges.

Application-generated high-rate continuous insertion and analytical workloads become read- and write-intensive to secondary storage hardware of data nodes. Additional access paths can reduce occurring unnecessary read effort and improve latencies, but require frequent maintenance. Whereas basic table storage management structures like Snapshot Isolation Append Storage (SIAS, compare Section 3.1.5.1) [Got16] obtain beneficial append-based write patterns very well, traditional strict alpha-numeric-sorted additional access paths result in a random write pattern, poor latency and high write amplification (WA, ratio between logically and physically written size of data). Additional access paths entail massive pressure, amplified by WA unbuffered to secondary storage. *Bandwidths of secondary storage devices are not efficiently used and shrink overall throughput of DBMS*. Maintenance costs of the additional access path become more expensive than scanning the base table. Considering different characteristics of main-memory primary storage and Flash-based secondary storage techniques, the problem space grows with complex memory hierarchies (compare in Sections 2.1 and A.1). *Generally, input-/output-operation (I/O) patterns of traditional additional access paths do not leverage characteristics of Flash-based secondary storage devices and shrink performance in read- and write-intensive workloads*.

By this means, company's investments are not optimally used. Furthermore, durability of secondary storage devices shrinks due to high update rates and WA of additional access paths. Endurance of Flash-based storage media depends on physical write-/erase-cycles. *Write patterns and WA of strictly key-sorted additional access paths drastically wear out secondary storage devices*.

A key characteristic of almost all commercial and academic DBMS is the application of Multi-Version Concurrency Control (MVCC, compare Section 2.3). Multiple tuple versions are maintained, valid for a different period in time. Major benefit is an improved throughput in the DBMS because concurrent reads and writes are not mutually blocking and proceed in their own calculated snapshot. Supplemental work for detection of a transactions visible tuple version is necessary. *Typically, additional access paths are version-oblivious due to maintenance costs* and the DBMS determines visibility by means of expensive base table look-ups or large cached in-memory structures. In principle, Flash-based secondary storage devices perform well on reads due to internal parallelism and asymmetric read/write behavior (outlined in Section 2.1), but visibility checking intensifies amount of – in principle – unnecessary read data, especially in an HTAP-scenario.

Briefly, DBMS and their characteristics are the linkage between application-generated workloads and the optimal usage of hardware technologies. Leveraging these characteristics can reduce running expenses of a business and allow new types of workloads. *Current additional access paths as well as virtually every storage manager do not provide adequate characteristics*. The goal of this work is to close this gap.

1.3. Contributions

Central contributions of this thesis are the conception, development and evaluation of a storage and index management structure, which meet the demand on modern workloads and hardware technologies. To the best of one's knowledge no other storage and index manager fully integrates these aspects in one single consistent structure with the result of substantial cuts in performance and endurance of data nodes, architectural fuzziness in DBMS or massive complexity in evolvement and adjustment of DBMS backend.

The presented approach is **Multi-Version Partitioned BTree** (MV-PBT), an unified tree-based multi-version storage and index management structure. Its origin and basis capabilities of a regular and well studied B^+-Tree [BM70] enable a simple integration in existing architectures and adaptability of recently published B-Tree techniques. Major benefit of tree structures is the natural alpha-numeric sort order, which enable – in contrast to hash-based indexes – powerful range querying in HTAP analytical workloads. In addition, due to the lack of pre-filtering data ranges, a range filtering approach is presented. **bloomRF** (**bloom r**ange **f**ilter) allows data skipping and increase throughput in MV-PBT by firstly introducing piecewise-monotone hash functions and prefix hashing in a bloom filter.

Research objectives are formulated on base of the contributing approach in following research questions (**RQ**):

RQ1: How could a visibility check of multi-version data be performed in Partitioned BTrees and leveraging modern hardware characteristics?

RQ2: What further applications arise from timestamp-based index-only visibility-checking in a MV-PBT?

RQ3: What optimizations for reading behavior are required for MV-PBT in the areas of data skipping and buffer efficiency?

RQ4: How do online reorganization methods in MV-PBT enable a workload adaptivity?

RQ5: What are the performance effects of optimized garbage collection in MV-PBT?

More and detailed contributing aspects of this thesis are as follows:

Formulation of findings, how a storage and index management structure need to interact, based on characterization of modern workloads and hardware technologies. In order to understand evaluation and conception of its linking layer – the DBMS storage structures – a brief characterization of modern workloads and hardware technologies is provided. Impacts on data management structures are elaborated. Thereby, the relevance of DBMS design decisions and underlying core mechanics become visible.

Theoretical and experimental evaluation of version models in multi-version DBMS and K/V-Stores. Primarily, the concepts and impacts of Multi-Version Concurrency Control (MVCC) designs, buffer management and additional access paths are focused on. One key finding in DBMS is the contrary optimal version organization model in base table main storage and additional access path maintenance.

Analysis of state-of-the-art literature, as well as theoretical and experimental evaluation of existing storage and indexing approaches. This thesis outlines a brief review of current state-of-the-art storage and index management structures in the areas of multi-version capabilities and leveraging modern storage hardware.

Introduction of Index-Only Visibility Check and demonstration of its relevance, especially in an HTAP scenario. Visibility checking is an expensive operation with linear growth to the length of version chains, but is required in multi-version DBMS. Modern indexing approaches require to support this operation.

Enabling a strict concept of out-of-place update and invalidation. MV-PBT introduce several different record types, whereby robustness in high concurrency and update-intensity situations is enabled.

Conception and evaluation of workload adaptivity, reorganization and garbage collection techniques in MV-PBT. Whereas state-of-the-art storage management structures focus on compaction mechanics for increased look-up performance by reduction of read-amplification (RA), MV-PBT aims to achieve a near-optimal WA and leverage its natural batch-wise partitioning sequence in order to guarantee robust access and update performance.

1.4. Scientific Publications

Contributions of this thesis and influential extensive work have been introduced in various scientific peer-reviewed or preprint publications, denoted in following list:

[MRBP23] B. Moessner, C. Riegger, A. Bernhardt, I. Petrov. 'bloomRF: On Performing Range-Queries in Bloom-Filters with Piecewise-Monotone Hash Functions and Prefix Hashing'. In: EDBT'23 (accepted) (2023)

[RP22] C. Riegger, I. Petrov. 'Storage Management with Multi-Version Partitioned BTrees'. In: ADBIS'22 (accepted) (2022).

[RBMP20] C. Riegger, A. Bernhardt, B. Moessner, I. Petrov. 'bloomRF: On Performing Range-Queries with Bloom-Filters based on Piecewise-Monotone Hash Functions and Dyadic Trace-Trees'. In: CoRR abs/2012.
15596 (2020). arXiv: 2012.15596. url: https://arxiv.org/abs/2012.
15596

[RVGP20] C. Riegger, T. Vinçon, R. Gottstein, I. Petrov. 'MV-PBT: Multi-Version Index for Large Datasets and HTAP Workloads'. In: Proceedings of the 23rd International Conference on Extending Database Technology (EDBT 2020). Copenhagen, Denmark, 2020.

[VWB+20] T. Vinçon, L. Weber, A. Bernhardt, A. Koch, I. Petrov, C. Knödler, S. Hardock, S. Tamimi, C. Riegger. 'nKV in Action: Accelerating KV-Stores on NativeComputational Storage with Near-Data Processing'. In: Proc. VLDB Endow. 13.12 (2020), pp. 2981–2984.

[VHR+19] T. Vinçon, S. Hardock, C. Riegger, A. Koch, I. Petrov. 'nativeNDP: Processing Big Data Analytics on Native Storage Nodes'. In: ADBIS. 2019

[PKH+19] I. Petrov, A. Koch, S. Hardock, T. Vinçon, C. Riegger. ‘Native Storage Techniques for Data Management’. In: 35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019. IEEE, 2019, pp. 2048–2051.

[RVP19] C. Riegger, T. Vinçon, I. Petrov. ‘Indexing Large Updatable Datasets in Multi-Version Database Management Systems’. In: Proceedings of the 23rd International Database Applications & Engineering Symposium. IDEAS ’19. Athens, Greece: Association for Computing Machinery, 2019.

[RVP18a] C. Riegger, T. Vinçon, I. Petrov. ‘Efficient Data and Indexing Structure for Blockchains in Enterprise Systems’. In: Proceedings of the 20th International Conference on Information Integration and Web-Based Applications & Services. iiWAS2018. Yogyakarta, Indonesia: Association for Computing Machinery, 2018, pp. 173–182.

[RVP18b] C. Riegger, T. Vinçon, I. Petrov. ‘Efficient Data and Indexing Structure for Blockchains in Enterprise Systems’. In: IBM Technical Report RC25681. 2018

[PVK+19] I. Petrov, T. Vinçon, A. Koch, J. Oppermann, S. Hardock, C. Riegger. ‘Active Storage’. In: Encyclopedia of Big Data Technologies. 2019

[PKV+19] I. Petrov, A. Koch, T. Vinçon, S. Hardock, C. Riegger. ‘Hardware-Assisted Transaction Processing: NVM’. In: Encyclopedia of Big Data Technologies. 2019

[VHR+18] T. Vinçon, S. Hardock, C. Riegger, J. Oppermann, A. Koch, I. Petrov. ‘NoFTL-KV: TacklingWrite-Amplification on KV-Stores with Native Storage Management’. In: Proceedings of the 21th International Conference on Extending Database Technology (EDBT 2018). Vienna, Austria: OpenProceedings, 2018, pp. 457–460

[RVP17a] C. Riegger, T. Vinçon, I. Petrov. ‘Multi-Version Indexing and Modern Hardware Technologies: A Survey of Present Indexing Approaches’. In: Proceedings of the 19th International Conference on Information Integration and Web-Based Applications & Services. iiWAS ’17. Salzburg, Austria: Association for Computing Machinery, 2017, pp. 266–275.

[RVP17b] C. Riegger, T. Vinçon, I. Petrov. ‘Write-Optimized Indexing with Partitioned B-Trees’. In: Proceedings of the 19th International Conference on Information Integration and Web-Based Applications & Services. iiWAS ’17. Salzburg, Austria: Association for Computing Machinery, 2017, pp. 296–300.

1.5. Structure of this Thesis

Since the motivational context, consequential problem statements and contributions are defined, the residual thesis is structured as follows. In Chapter 2, a brief introduction in the technical background is given. Thereby, required insights and concepts in the areas of *hardware technology characteristics*, *workload characteristics* and *fundamentals of DBMS designs* are provided. Based on the outlined findings, several mostly key-sorted indexing approaches are introduced and evaluated in Chapter 3. The designated founding approach Partitioned BTrees (PBT) is elaborated and enhanced to version-aware Multi-Version Partitioned BTree (MV-PBT), coping determined concepts and capabilities in Chapter 4. Shortcomings in existing point-range filter techniques – a basic requirement for efficient data skipping in MV-PBT – lead to the concept and development of bloomRF outlined in Chapter 5. Chapter 6 provides a full integration and experimental evaluation of MV-PBT in an append-based DBMS and K/V-Store. Finally, a brief summarization and opportunity overview is presented in Chapter 7.

Chapter 2

Technical Background

With the aim of understanding the various aspects and challenges of key-sorted access paths, modern complex memory hierarchies and diverse characteristics of its members are described first. Subsequently, natural behavior of evolving workload types are outlined. Thereupon, fundamentals of modern database management system (DBMS) design techniques are provided. With respect to formulated characterizing statements, an emerging significance in robust and Flash-leveraging key-sorted multi-version storage and index management structures is elaborated.

2.1. Characteristics of modern Hardware Technologies

Understanding basic concepts and characteristics of modern hardware technologies is crucial for database access method design. Execution time on hardware can be simplified to processing ($T_{processing}$) and data access ($T_{data-access}$) costs. Adopting trends towards resource and memory efficiency in modern K/V-Stores [EGA+18; Inc22; MWKM15], access paths must consider characteristics of secondary storage devices. Even though, instructions in data-intensive operations are frequently performed, access latencies to massive amounts of data on high-capacity secondary storage are assumed to become a dominant factor, so that:

$$T_{data-access} \gg T_{processing} \tag{2.1}$$

Therefore, the scope of this work is focused on characteristics of the memory hierarchy. Due to the reliance of processing and data provisioning, a quite brief characterization of recent processing evolutions is given.

2.1.1. Highly parallelized and decentralized processing units

Over the past decades, processing power benefits from increasing clock speeds. Applications generally profit by this trend. Latest developments in processing units yield increased parallelism and manifold unconventional calculation design options. Even though the scope of this thesis is not primarily focused on optimizing processing costs, one central point can be formulated: *Data access methods require to be aware of decentralized computing models*. Multi-core central processing units (CPU) and various possible hardware accelerator units are probably involved in modern DBMS [BGHS19] and require to complementary operate on a consistent dataset instance, without shrinking performance due to concurrency issues, like excessive cache-invalidation or double buffering. Efficient data provisioning and persistence for decentralized and parallel calculations is the purpose of memory hierarchies on data node servers.

2.1.2. Primary and Secondary Storage Characteristics

Components of the memory hierarchy are the bottleneck of massively data-intensive tasks. Efficiently using clock-cycles of processing units rely on data provisioning along these volatile primary and persistent secondary storage devices.

Latency describes the delay in response time between components. As a general rule, latencies, but also capacities, increase with higher distance to a processing unit. CPU caches retrieve cache-line-sized data from main-memory (e.g. RAM). Whenever an already persistent (mostly volatile) working copy is not present on a byte-addressable memory device, it is retrieved by a *read I/O* operation from a block-addressable storage device. The other way around, in volatile memory newly allocated data or modified working copies are persisted on block-addressable secondary storage devices by *write I/O* operations.

Traditionally, in DBMS algorithms, only two levels in memory hierarchy are considered – volatile main-memory and and persistent disk [Gra11]. Latencies of read and write I/O operations form a massive *access gap* (compare Section A.1). Whereas wait latencies between caches and memory is about $\times 10^2$, the access gap between caches and HDD (traditional cheap mechanic Hard Disk Drive) is about $\times 10^6$. Common behavior in traditional memory and storage media is formed in symmetric read and write latencies as well as benefits in sequential operations.

Evolving and recently established storage technologies close the access gap (compare *Complex Memory Hierarchies* in Section A.1), but also broaden the characteristics of devices in memory hierarchies. Reflecting latencies of solid state drives (SSD) and non-volatile memories (NVM), they naturally fill the gap, however, their special characteristics[1] must be considered:

High level of inherent parallelism. Independent or decomposed execution of I/O operations within several structural levels of an SSD has been well

[1]Characteristics are exemplary taken from SSDs. A detailed technical background is given in Section A.2.

studied [APW+08; HJF+11; PSS+10; RZA+12; Shi17; SXX+09; WKS15]. Leveraging structural characteristics by such techniques enable much higher parallelism and I/O-performance as has been known from HDD.

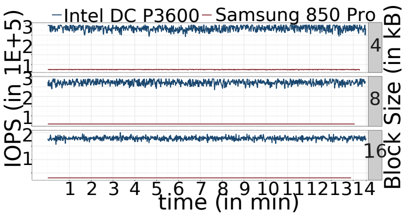
Access structures might leverage high parallelism whilst accessing data – i.e. SSDs cope with more purposeful I/O of DBMS and K/V-Stores.

Asymmetric read and write performance. Flash mainly supports the native operations *read*, *write* and background *erase*. Reads perform an order of magnitude better than writes and two orders of magnitude better than erases [CKZ09; Got16] regarding latencies and IOPS (I/O per second). Especially in case of random write I/O, the latency and costs alternate between cheap sequential write and expensive erase operations [BJB09]. SSDs contain several on-device caches for many purposes: caching of address translation mapping tables, performing asynchronous read ahead, concealing asymmetric behavior by write buffering or enabling background tasks. Generally, operation costs depend on the internal circumstance of an SSD. High-rate continuous I/O operations uncover asymmetric behavior in steady-state conditions [DISK20; Got16]. As depicted in Figure 2.1b, write throughput quickly drop as caches satiate at different utilization levels by block sizes. In contrast, asymmetric read behavior remains constant in all block sizes (Figure 2.1a).

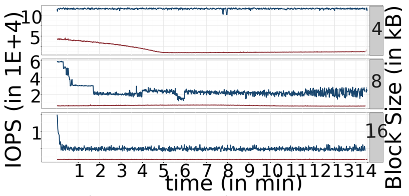
Access structures leverage asymmetric I/O characteristics by minimizing write accesses, whereas SSDs cope with increased read I/O.

Out-of-place update operation. Unlike to HDDs, in which a page at a specific address can be physically overwritten in-place, Flash require a clean page status before writing. Rewriting a page requires an out-of-place write at a different clean page and an update of the address translation mapping table. The costly erase is performed by a background garbage collection (GC) task at a later point in time. [CKZ09; DISK20]

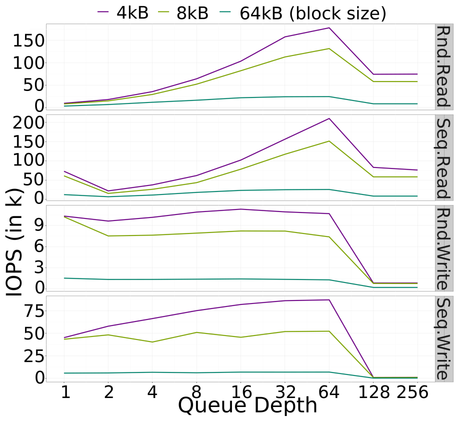
Access structures must avoid in-place updates to already persisted data and replace modifications by out-of-place updates.



(a) IOPS Random Read



(b) IOPS Random Write



(c) Intel DC P3600 - Sequential vs. Random

Figure 2.1.: Read and write metrics of *Intel DC P3600* enterprise SSD and *Samsung 850/860 Pro* consumer SSD applied in the testbed.

Advantages in sequential write I/O pattern. As an effect of out-of-place page replacement, inherent parallelism and low memory footprint hybrid-/block-level-mapping in on-device caches, sequential write patterns can improve performance of an SSD [MFL14]. Sequential sectors can be arranged and distributed along independent Flash components. Random writes do not result in sequential sectors, whereby memory footprint of mapping tables increase and background maintenance operations, like garbage collection, become more complex and expensive. As a general rule, "*random writes should be limited to a focused area*" and "*sequential writes should be limited to a few [concurrent] partitions*" [BJB09]. Figure 2.1c indicates roughly equal very good performance for sequential and random read I/O for different page sizes and numbers of pending I/O requests (queue depth), however, random write performance is almost a fraction of the sequential operation for all measurement points.

Generally, access structures must adapt sequential writes for vast majority of payload, whereas few random writes of small meta structure data are feasible. Contrary, read latencies roughly remain unaffected by access patterns.

Background Garbage Collection (GC). Writing a page requires a clean on-device target-page status. As a result, write-back operations of updated pages are performed at a different storage location (out-of-place) and the former page version becomes invalid. Erase operations are very expensive but are required for space reclamation. A fundamental property in Flash (NAND) is, that data can be read and written on page level, but require to be deleted on superior block level. First, a victim block is determined. Second, still valid pages in the block are written to an appropriate different block location. Third, the victim block becomes erased. As a side effect, the write amplification (WA) increases on device level (WA-D). [MFL14; MWKM15]

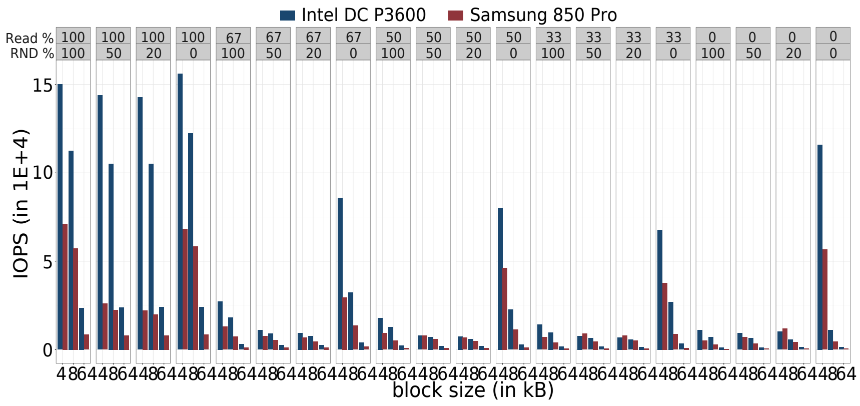
Access structures must consider GC to be valuable for larger regions. Temporary space amplification by obsolete data is negligible on secondary storage.

Limited durability and wear. Status of memory cells in pages and blocks is switched by write and erase operations in a write-erase-cycle. Each write-erase-cycle in a block wear out the transistor oxide layer of memory cells. The approximate number of maximum write-erase-cycles depends on the underlying technique, whereas 3000 cycles are common for consumer devices and 10.000 for enterprise devices. SSDs track the number of erases for each block. Wear-leveling methods equally distribute wear over all blocks and consequently, unreliable blocks become discarded. [MFL14; MWKM15]

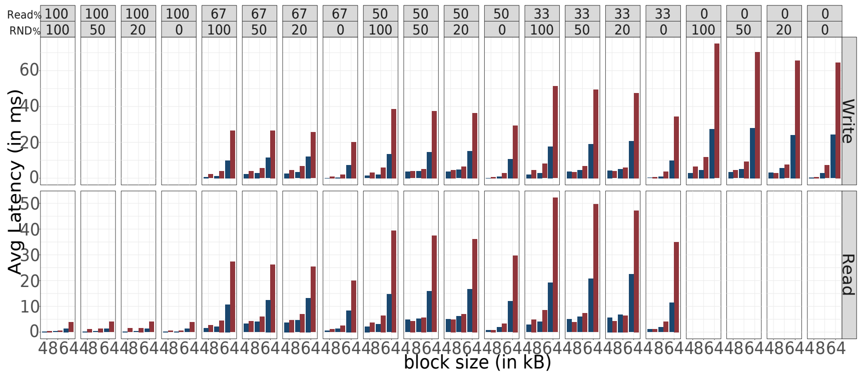
By knowledge of the maximum write-erase-cycles ($N_{w/e-cycles}$) per block, the absolute storage capacity of the SSD ($S_{capacity}$) in blocks, the modified data by a workload ($S_{modified}$) and the write amplification (WA of data structure and on-device) the durability (D) is calculated as follows:

$$D = \frac{N_{w/e-cycles} \times S_{capacity}}{WA \times S_{modified}} \tag{2.2}$$

In the denominator, the multiplication of WA and $S_{modified}$ defines the wear in written blocks. Since wear is equally distributed over all blocks, the absolute number of write-erase-cycles of the SSD is calculated by the multiplication of $N_{w/e-cycles}$ and $S_{capacity}$ in the numerator. *Access structures must minimize WA, since it negatively affects Flash's durability.*



(a) IOPS - random/sequential , read/write distribution



(b) Latency - random/sequential , read/write distribution

Figure 2.2.: IOPS and latency metrics of applied SSDs in the testbed for different workload characteristics and block sizes.

I/O transfer time is majority. Accessing data on secondary storage requires time for seek and transfer. Since SSDs have no electro-mechanic components, seek time is very low, whereas, unlike to HDD, transfer time becomes majority. Larger transfer / page sizes result in fewer I/O per second (IOPS, compare Figure 2.2) for different access patterns.

Access structures must consider to optimize net transferred data by cheap access structures and appropriate page sizes.

2.1.3. Testbed Hardware Metrics and Operating System

Performance evaluation benchmarking is performed on a 64-Bit *Ubuntu 16.04.7 LTS (xenial)* server setup with Linux Kernel Version *4.4.0-210-generic*. Benchmarks and DBMS are compiled with GCC Version 9.4.0.

The CPU is a 4 core (8 thread hyper-threading) *Intel(R) Xeon(R) CPU E5-1620 v3 @ 3.50GHz* with internal L1/L2/L3 256kB/1MB/10MB caches. Memory is provided by 2×16GB DDR4 RAM devices (manageable up to 32GB by boot parameters). The server is equipped with a 400 GB *Intel DC P3600* PCIe enterprise SSD and a 500GB and 1TB SATA *Samsung 850/860 Pro* consumer SSD, whereas logging and payload perform without any interference and on different intensity-levels. Generally, write caches are disabled for steady state, if possible. Read and write characteristics are depicted in Figures 2.1 and 2.2 and are in accordance to outlined Flash characteristics in Section 2.1.2.

2.2. Modern Workload Properties

In order to design, implement and evaluate a version-aware storage and index management structure that leverages modern hardware characteristics, several types of workloads are considered.

First, Online Transaction Processing (OLTP) represents traditional business contextual data management use cases, executing small updates and inserts alongside to multiple reads [TPC10].

The very opposite in characteristics of query execution is a decision support Online Analytical Processing (OLAP) workload, whereby queries with high degree of complexity and large volume of examined data are performed [TPC21]. An upcoming type of workload is a mixture of OLTP and OLAP in Hybrid Transactional and Analytical Processing (HTAP) on a huge shared dataset instance [CFG+11; HG20; ÖTT17].

Cloud data serving workloads bring OLTP-like transaction processing and massive amounts of data and high-rate continuous insertions together [CST+10].

Some general assumptions about modern workload properties are as follows:

Exponential growth of data. Ongoing digital transformation and personalization of services incur rapidly growing dataset sizes [DB20; SG06]. Massive amounts of data require to be efficiently managed.

High-rate continuous insertion / modification. Evolving types of applications and data sources with varying lifespan incur frequent data modification. Furthermore, availability and quality of services require stable steady state throughput (elasticity) of high-rate continuous insertion and modification workloads [CST+10].

Application of HTAP workloads. Integration of data relying on various isolated solutions lead to inflexible and complex infrastructure landscapes. Data-centric infrastructures, whereby applications and services are designed and built upon a core of commonly applied and maintained dataset instance [DB20; HG20; ÖTT17; RVGP20], enable cheap administration and flexible evolvability.

2.2.1. Application of standardized Database Benchmarks

Introduced workload property assumptions include high-rate continuous insertions and modifications with simultaneous analytical querying on massive amounts of data, which potentially are performed on a shared dataset instance. Properties are well represented in HTAP as well as cloud serving benchmarks.

It is common practice to apply a best concurring standardized database benchmark. Popular benchmarks are designed by the Transaction Processing Performance Council (TPC)[1].

[1]'*The TPC is a non-profit corporation focused on developing data-centric benchmark standards and disseminating objective, verifiable data to the industry.*' [TPC22]

The TPC-C OLTP benchmark is very common in database benchmarking. OLTP mainly performs transactions of multiple reads as well as small updates and inserts [TPC10] in a business contextual database schema with multiple indexes (compare Figure 2.3 TPC-C). The open source DBT-2 [WW21] implementation has been applied as business contextual OLTP benchmark in this thesis.

Offside the commonness of TPC-C as OLTP benchmark, however, there is a significant reason for its application. The authors of [CFG+11] recognized the lack of meaningful HTAP workloads and developed the CH-Benchmark upon TPC-C (TPC OLTP-workload) and TPC-H (TPC OLAP-workload). As depicted in Figure 2.3, CH-Benchmark operates on a commonly shared dataset instance, yielding special HTAP characteristics. This benchmark is integrated in the OLTP-Bench framework [Pav+21].

As an alternative for high-rate continuous insertion workload on K/V-Stores, the *Yahoo! Cloud Serving Benchmark* (YCSB) [CST+10] is applied in [RKD21]. YCSB consist of different key-value style workloads for cloud-based applications. In the core workload package, several mixtures of key-value put-, get- and scan-operations for different data and request distributions as

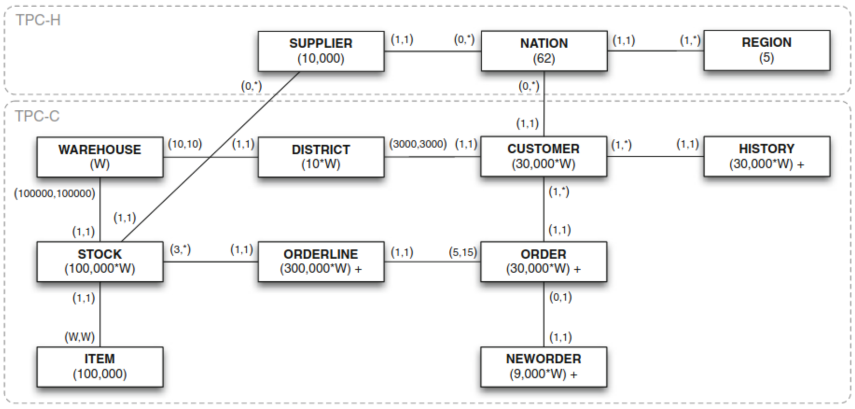


Figure 2.3.: Schema of the CH-Benchmark [PWM+14] adopts TPC-C and TPC-H as an HTAP workload representative.

well as sizes are supported. Many parameters are adjustable for simulation of arbitrary use-cases.

2.3. Fundamental Design Decisions in Multi-Version Database Management Systems

Multi-Version Database Management Systems (in this thesis simply denoted as DBMS) as well as K/V-Stores are the backbone of data-intensive applications. By this are meant all DBMS, which implement the Multi-Version Concurrency Control (MVCC) protocol and apply Snapshot Isolation (SI). Well-known MVCC is one of the most popular transaction management schemes with many representatives: Oracle [Ora21a], Microsoft SQL Server [Mic21b], HyPer [KN11], SAP HANA [FML+12], Google Cloud Spanner [BBB+17], MongoDB WiredTiger [Mon21], NuoDB [Nuo21], PostgreSQL [Pos21] or MySQL-InnoDB [Ora21c], just to name a few.

In Figure 2.4, modules of a DBMS are schematically depicted. Architectures in K/V-Stores vary and potentially enable less features and modules. Users interact with the DBMS by an user interface according to which the user request is parsed in an execution plan. Thereupon, the query is executed by accessing data with the best matching access method and perhaps performing further operations. A required set of transaction properties is guaranteed by the concurrency control module, namely MVCC. Accessible data in massively data-intensive operations is likely to be located on secondary storage devices, e.g. on Flash SSDs. Access methods request the byte-accessible data from a RAM-located Buffer Manager. If the data is not represented in the Buffer Manager, it is read from block-addressable secondary storage. In order to guarantee maximum memory footprint, buffers require to be evicted from cache by a replacement policy. Unchanged buffers can be discarded, modified ones require to be written to secondary storage media by logic of the Storage Manager. Data corruption is avoided by logging of critical operations.

Scope of this thesis is the storage and index management represented in Access Methods and accompanying modules Buffer and Storage Manager

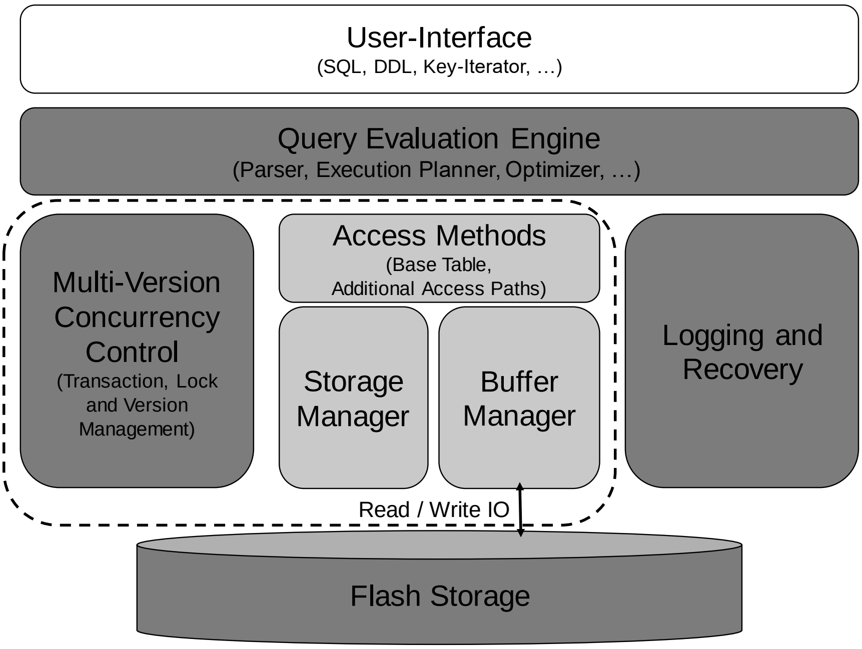

User-Interface
(SQL, DDL, Key-Iterator, …)
Query Evaluation Engine
(Parser, Execution Planner, Optimizer, …)
Multi-Version Concurrency Control
(Transaction, Lock and Version Management)
Access Methods
(Base Table, Additional Access Paths)
Storage Manager
Buffer Manager
Logging and Recovery
Read / Write IO
Flash Storage

Figure 2.4.: Regular DBMS Architecture (compare [RG03]). Scope of this thesis is in dashed lines, whereby critical interfering modules are shaded in darker and main contributions in brighter gray.

with focus on properties of interlinked MVCC and beneficial I/O to secondary storage devices. Query Evaluation Engine as well as Logging and Recovery modules are affected as well, though are not treated in detail. Therefore, a detailed overview of MVCC designs is given.

2.3.1. Opportunities and Version Management in Multi-Version Concurrency Control

Multi-Version Concurrency Control (MVCC) and Snapshot Isolation (SI) are well-known mechanisms in DBMS. However, fundamentals of MVCC with SI are essential prerequisites in this thesis, therefore some background is provided.

Parallel execution of transactions is a major scaling factor of DBMS, whereas throughput and response times are improved, however, interleaving concurrent transactions can cause non-tolerable data anomalies to an application. Several well-known version-oblivious concurrency control mechanisms enable integrity, nevertheless typically reading and writing transactions are mutually blocking, i.e. readers require to wait for completion of a writer and writers wait for lock releases of readers. In other words, this means in an HTAP scenario that OLAP queries on update-intensive data never complete and OLTP modifications are not able to proceed. MVCC with SI overcome these issues.

MVCC creates a new *version* of a data tuple on each update. As a result, readers do not block writers anymore, because read locks do not oblique writers to wait for release. In MVCC, it is very common to leverage its versioning behavior in *Snapshot Isolation (SI)*. SI let transactions operate on a consistent (logical) state (snapshot) of the database, e.g. the latest committed change to a tuple at begin of the transaction, respectively its own modification, is visible. Moreover, SI prevents concurrent transactions from updating the same tuple. Due to the fact, that every transaction knows its snapshot, writers do not block readers anymore. MVCC with SI enable high concurrency and isolation level – even in HTAP workloads with long-lasting

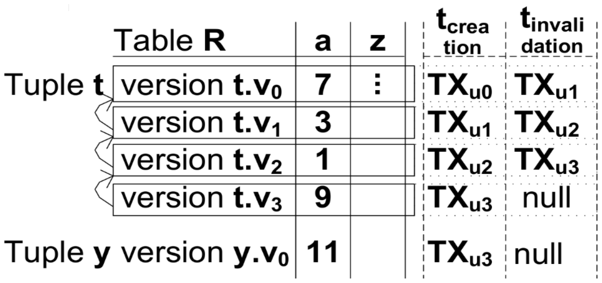


Table R
a
z
t_creation
t_invalidation
Tuple t
version t.v0 | 7 | ⋮ | TXu0 | TXu1
version t.v1 | 3 | TXu1 | TXu2
version t.v2 | 1 | TXu2 | TXu3
version t.v3 | 9 | TXu3 | null
Tuple y version y.v0 | 11 | TXu3 | null

Figure 2.5.: Logical View on Tuple Versions in MVCC with creation and invalidation timestamps [RVGP20]. Physical order typically differ.

analytical queries and frequent updates. *Therefore, it is a good choice to leverage parallelism in processing units and Flash storage.*

From a logical point of view, maintenance of tuple versions bases on some prerequisites. Each tuple consists of at least one *tuple version record*. By this means, several tuple version records of one logical tuple exist in a *version chain* and each of them is valid for a different period in time. Validity of a tuple version, strictly speaking its visibility to a specific transaction snapshot, is determined by timestamps for creation ($t_{creation}$) and invalidation ($t_{invalidation}$)[1]. A *visibility check* is responsible to return the valid tuple version to a transaction snapshot, whereby the version chain is processed and the timestamps are evaluated. In order to process the version chain, an *entry point* to the chain must be known, i.e. a specific tuple version record, and each tuple version requires to know its predecessor or successor. Version chains preferably have the structure of a latch-free singly linked list [WAL+17]. Tuple version records become obsolete, if it is no more visible to any active transaction snapshot, and is removed by *garbage collection (GC)*.

Figure 2.5 depicts a logical view on a tuple version chain. Tuple t was changed by transactions ($TX_{u0}, TX_{u1}, TX_{u2}, TX_{u3}$ several times by modifying the attribute a to $7, 3, 1, 9$. Each transaction maintains timestamps for creation and appropriate invalidation of its predecessor at the logical tuple version. Tuple y depicts a tuple version, which is not a member of the mentioned tuple version chain of tuple t. *The depicted creation of new tuple version records is in accordance to out-of-place update characteristics of Flash secondary storage devices.*

Considering creation of a new version each time a tuple is updated in MVCC, this might happen in different ways – detailedly outlined in the following sections. Several DBMS implement variations of the version management scheme, whereby the design decisions specify its capabilities and applicability. [WAL+17] evaluated different designs from a main-memory

[1]Transaction timestamps constitute a logical sequence of executed transactions. In addition, DBMS possibly maintain further information, e.g. a sequence of command numbers per transaction in PostgreSQL [Pos21], in order to avoid race conditions. In this thesis, transaction timestamps are meant to consider command numbers in the logical sequence as well.

point of view. Building up on these insights, different MVCC designs require to be evaluated in this thesis with respect to characteristics of secondary storage devices in a complex memory hierarchy [RVGP20].

2.3.1.1. Version Storage

A logical tuple corresponds to one or more tuple versions, which form a singly linked list of records in a version chain (Figure 2.5). There are two possible physical representations of a tuple version (Figure 2.6): *physically materialized* or *delta-record-based*. The *former* implies that each tuple version record is entirely stored physically materialized. By this means, each tuple version record contains all valid information of a logical tuple in a specific period of time. Whenever a version record is valid to a transaction snapshot, further records are not required to restore the logical tuple. The *latter* implies that each modification of a logical tuple results in a delta record, which indicates the difference to another tuple version (e.g. applied in BW-Tree [LLS13; WPL+18]). Delta records are linked and require to be retrieved on demand by the DBMS storage manager for tuple reconstruction. Delta-record-based system designs typically store a single version record (oldest or newest – depends on version ordering, compare Section 2.3.1.2) in the main store. Delta records are located in a separate storage location, e.g. the undo log (applied in InnoDB [Ora21c]) or a temporary version record store (applied as option in MS SQL Server [Mic21b]). By this means, only the version record in the main store can be directly accessed, other

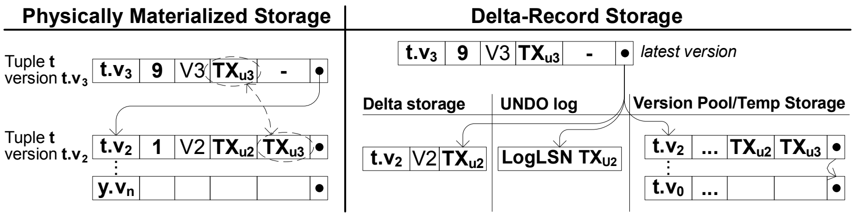

Physically Materialized Storage
Delta-Record Storage
Tuple t version t.v3
t.v3 | 9 | V3 | TXu3 | - | ●
Tuple t version t.v2
t.v2 | 1 | V2 | TXu2 | TXu3 | ●
y.vn | ●
t.v3 | 9 | V3 | TXu3 | - | ●
latest version
Delta storage
UNDO log
Version Pool/Temp Storage
t.v2 | V2 | TXu2
LogLSN TXU2
t.v2 | ... | TXu2 | TXu3 | ●
t.v0 | ... | ●

Figure 2.6.: Version Storage Alternatives [RVGP20]

version records require to retrieve the required information by processing the main version record and all intermediate delta records up to the valid tuple version is fully reconstructed.

Both physical representation models are capable to perform modifications in-place or out-of-place. The former creates a copy (or delta) of the most recent valid tuple version representation at some other location and maintains version chain linkage information, transaction timestamps and desired modifications to the newer version in-place. The latter let the most recent valid tuple version representation unchanged and creates a tuple version record (or delta record) with desired modifications at some other location and maintains version chain linkage information and transaction timestamps.

Considering the characteristics of modern Flash storage (outlined in Section 2.1.2), physically materialized version record storage and out-of-place update scheme are preferable, due to lower RA, WA and tuple reconstruction costs. Delta records tend to consume less space than materialized tuple versions, especially in case of large tuple information sizes and small modifications, but require additional processing and all predecessors or successors for tuple reconstruction.

2.3.1.2. Version Ordering

Tuple version records of a logical tuple form a tuple version chain. Its organization of a singly linked list enables lock-free modifications. As a

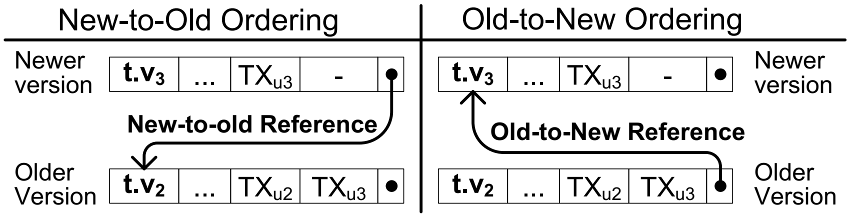


Figure 2.7.: Version Ordering Alternatives [RVGP20]

result, there is one *entry point* to the tuple version chain. Every tuple version record is accessible by successively processing the tuple version chain – beginning with the entry point. Internal ordering in this singly linked list can be organized from old-to-new or new-to-old (depicted in Figure 2.7). The former organization enable a static reference to its entry point. A predecessor require to know its successor, wherefore references in the predecessor must be modified. Adding a new tuple version record requires to process the whole tuple version chain beginning from the entry point and append it to the end of the list. The latter organization model, namely new-to-old, prepends newly inserted tuple version records to the entry point of the tuple version chain, whereas it becomes the new entry point. The new record references the old entry point, which requires no modifications.

Treading mixed workload characteristics like HTAP, both version ordering schemes favor different types of queries. Old-to-new ordering scheme tend to favor long-lasting OLAP queries, precisely because visible tuple version records to its transaction snapshot are located near the entry point. Conversely, recently inserted tuple version records are much faster accessible in a new-to-old ordering scheme.

Considering the characteristics of modern Flash storage (outlined in Section 2.1.2]), new-to-old ordering scheme makes a leading edge, due to its version chain maintenance. Predecessor version records need not to update linkage information, contributing to reduce WA. A desirable append-only behavior is the case if combining out-of-place physically materialized tuple version records (compare Section 2.3.1.1) and outlined new-to-old ordering. Other combinations require in-place updates.

2.3.1.3. Version Invalidation Model

A tuple version is said to be invalidated whenever a successor version record exists. Two different invalidation models are considered [Got16] (Figure 2.8). *Two-point invalidation* is a widespread model, where the creation timestamp of the successor version is also placed as invalidation timestamp on the predecessor. By evaluation of a single tuple version record, its visibility to

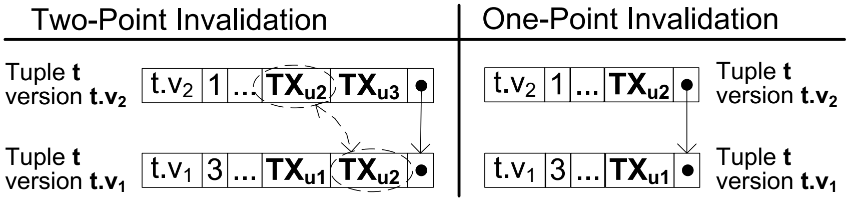

Two-Point Invalidation
One-Point Invalidation
Tuple **t** version **t.v₂**
Tuple **t** version **t.v₁**
t.v₂ | 1 | ... | TXu2 | TXu3
t.v₁ | 3 | ... | TXu1 | TXu2
t.v₂ | 1 | ... | TXu2
t.v₁ | 3 | ... | TXu1

Figure 2.8.: Version Invalidation Alternatives [RVGP20]

a transaction snapshot can be determined. On the other hand, the existence of a successor version itself implicitly invalidates its predecessor. *One-point invalidation* [GPHB17] leverages this point by maintaining only the creation timestamp at each tuple version record and implicit invalidation. Major benefit of this technique is the avoidance of in-place invalidation – what is actually a modification – at predecessors of tuple version records. However, evaluating the visibility of a tuple version record to a transaction snapshot probably requires successor information. One-point invalidation matches well previously outlined new-to-old ordering scheme, since all successors and respective creation timestamps are processed as part of the tuple version record discovery.

With regards to characteristics of modern Flash storage (outlined in Section 2.1.2), only one-point invalidation is capable to avoid in-place updates and therefore enables beneficial append-only behavior with low WA. Anyways, required successor versions are processed in case of new-to-old ordering scheme (compare Section 2.3.1.2), and therefore it is not considered to be any drawback.

2.3.1.4. Garbage Collection

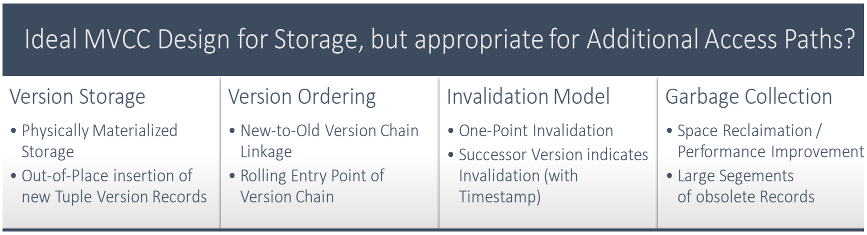
Modifications to a tuple result in the creation of a new tuple versions. Tuple version records become obsolete, if they are no longer visible to any of the active transaction snapshots.

Obsolete tuple version records require to be garbage collected (GC) for space reclamation and probably performance gains. However, GC reduces concurrency as some form of locking is required, causes performance spikes as it interferes with foreground I/O and increases WA on secondary storage devices. GC can be performed on transaction, tuple and index levels [LLS13; LSP+16; WAL+17; WPL+18].

In order to keep GC costs low, operations require to be worth it and should not break with append-only behavior and minimizing WA principles (compare Section 2.1.2). Furthermore, independence of access structures – i.e. base tables, additional access paths and potential helper structures – reduces locking effort and bouncing across structures.

2.3.1.5. Discussion

Different possible design decisions in MVCC are introduced and theoretically evaluated on aspects of modern secondary storage characteristics. As depicted in Table 2.1, MVCC designs can be combined to an optimal set for base table storage on Flash. An append-only new-to-old ordered storage manager with one-point invalidation for base tables is *Snapshot Isolation Append Storage (SIAS)* [Got16; GPHB17] (outlined in Section 3.1.5.1). SIAS exhibits in a write-heavy transactional workload 97% reduced write I/O and an improved throughput of 30% compared to its baseline MVCC with SI in PostgeSQL [GPHB17]. However, additional access paths become more complex as outlined in Section 2.4.



Ideal MVCC Design for Storage, but appropriate for Additional Access Paths?

Version Storage	Version Ordering	Invalidation Model	Garbage Collection
• Physically Materialized Storage • Out-of-Place insertion of new Tuple Version Records	• New-to-Old Version Chain Linkage • Rolling Entry Point of Version Chain	• One-Point Invalidation • Successor Version indicates Invalidation (with Timestamp)	• Space Reclaimation / Performance Improvement • Large Segements of obsolete Records

Table 2.1.: Optimal MVCC design for characteristics of Flash storage devices.

2.3.2. Estimated Base Table Cost Model

In the previous section a strict append-only heap organization is outlined, which leverages characteristics of Flash (compare Section 2.1.2). Major benefits of this base table organization are the minimal insert, update as well as delete costs (i_{SIAS}) and yielding WA_{SIAS} per page of strict append-only storage.

$$WA_{SIAS} \ = \ 1 \tag{2.3}$$

Since records are invalidated out-of-place, all modifying operations have the processing costs $i_{SIAS} = \mathrm{O}\ (\ 1\ )$ practically no read I/O (except for potential *entry point* identification) and optimal write I/O costs ($W_{I/O}$) per operation ($i_{I/O_{SIAS}}$)

$$i_{I/O_{SIAS}} = W_{I/O} \ \times \ \frac{B_{N+R}}{N+R} \tag{2.4}$$

whereby N comprise the set of logical tuples, R the totality of logical replacements and B_{N+R} the number of required pages (**b**uffers) to store all version records ($N+R$).

Search costs (s_{SIAS}), however, in heaps are much more expensive. Generally, search costs are declared in average as $s = \mathrm{O}\ (\ N/2\ )$, since the required record might be located anywhere in the dataset. However, version records are invalidated out-of-place, whereas principally the entire dataset must be scanned to identify the required tuple version record, hence $s_{SIAS} = \ \mathrm{O}\ (N+R)$ and yielding read I/O costs ($R_{I/O}$) per search as well as scan ($s_{I/O_{SIAS}}$) are defined as follows:

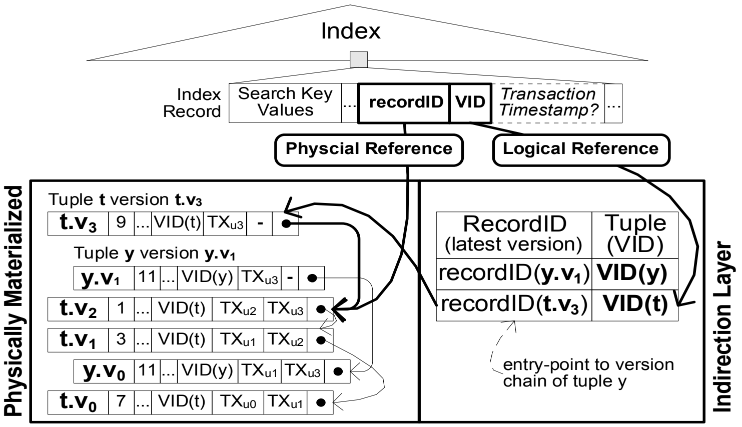
$$s_{I/O_{SIAS}} = R_{I/O} \ \times \ B_{N+R} \ \times \ (1-p_{Bc}) \quad , \ p_{Bc} \ \simeq \ 0 \tag{2.5}$$

Due to assumed data-intensive workloads and trends towards resource and memory efficiency (compare Sections 2.1, 2.2), a cache probability is assumed to be $p_{Bc} \ \simeq \ 0$, whereas each required page B_{N+R} cause $R_{I/O}$. By this means, cache probability per iterated tuple version (including $N+R$) is probably better in sequential processing and depends on the version records per page $\frac{N+R}{B_{N+R}}$.

2.4. Additional Access Paths in Multi-Version DBMS

DBMS objects organize data in tuple records and arrange them for optimal support of one or a few operations [RG03]. Different organization models, e.g. heap, sorted or hashed, are favorable for a set of operations, due to different cost models. A preferable base table organization for Flash storage characteristics might be a strict append-only heap. Thus, insertion operations in the base table heap are very cheap – as tuple records are collected in main-memory pages and get evicted once (as applied in SIAS [Got16]), however, equality and range search operations become very expensive (compare Section 2.3.2). *As a result, equality search costs increase to a scan of the whole multi-versioned dataset, as its size grows by the number of version records and the necessity of a visibility check*.

A fully attribute-key-sorted organization model probably improve performance of equality and range search operations, but could also break the



Index
Index Record
Search Key Values
...
recordID
VID
Transaction Timestamp?
...
Physcial Reference
Logical Reference
Physically Materialized
Tuple t version t.v3
t.v3 | 9 | ... | VID(t) | TXu3 | -
Tuple y version y.v1
y.v1 | 11 | ... | VID(y) | TXu3 | -
t.v2 | 1 | ... | VID(t) | TXu2 | TXu3
t.v1 | 3 | ... | VID(t) | TXu1 | TXu2
y.v0 | 11 | ... | VID(y) | TXu1 | TXu3
t.v0 | 7 | ... | VID(t) | TXu0 | TXu1
RecordID (latest version)
Tuple (VID)
recordID(y.v1) | VID(y)
recordID(t.v3) | VID(t)
entry-point to version chain of tuple y
Indirection Layer

Figure 2.9.: Version / Index Record Referencing [RVGP20]

desired append-only behavior. Furthermore, any equality or range search on a different attribute of the tuple result in a scan of the whole dataset. DBMS provide the option to create *additional access paths* (alias *indexes*) to base table primary data storage. Application of the auxiliary structures is manyfold, e.g. support of uniqueness constraint checks, primary keys and foreign keys or potentially fast look-ups on any indexed set of attributes (secondary index). Index structures apply a search optimized hashed or sorted organization model, however, modifications tend to become expensive, due to maintenance costs and resulting unfavorable random write I/O pattern to secondary storage devices. *Sorted indexes allow equality and range search with low RA*[1] *and avoidance of expensive downstream sort operations – unlike to hash indexes which allow only equality search, wherefore they are not suitable for HTAP scans and not in scope of this thesis.*

Index records regularly consist of a pair of *search key value(s)* and a one *data tuple reference*[2] [RG03], whereas they are version-oblivious, due to missing *transaction timestamp* information (compare Figure 2.9). The compact representation allows cheap search and maintenance costs, however, there is no guarantee for improved throughput of the DBMS, as any applied index requires maintenance. Administrators must evaluate the benefit of an index. Based on query attributes and statistics like data selectivity, the optimizer of the query evaluation engine (compare DBMS architecture in Figure 2.4) can decide to use the additional access path via index.

Since the index record is located fast, its referenced data tuple location in the base table is also known and can be accessed in a following step. A very classical way of data tuple referencing in an index record is a physical reference by *record id*. In the context of MVCC, several tuple version records of one logical data tuple exist. Principally, each tuple version record is required to be indexed, albeit it is sufficient for reliability to locate each matching tuple version record via the version chain. A rolling *entry point* in the new-to-old ordered version chain (as referred in Table 2.1) requires an

[1]Comparing linear heap table search costs of Equation 2.5 in Section 2.3.2 with exemplary logarithmic B^+-Tree costs of Equation 3.4 in Section 3.1.1.

[2]Reference organization variants, e.g. lists of data tuple references, are not considered.

updated physical reference for each modification – i.e. respectively in case of appended successor versions, physical record movement or performed garbage collection. Modifications to indexed attribute values become much more complex, as a new index record is inserted and the entry point reference of the predecessor index record requires to be fixed.

In order to avoid expensive index maintenance operations, some DBMS [DFI+13; Pos21] sacrifice base table characteristics from Table 2.1 and employ an old-to-new version ordering with in-place updates, in which case the entry point remains stable. A second way is to implement an indirection layer between index and base table. Each data tuple record is augmented with an unique tuple identifier (Virtual Tuple Identifier – VID), which is stored in the index record reference pointer. An index operation resolves the VID via usage of a mapping table in order to locate the physical entry point of a data tuple (as depicted in Figure 2.9). An indirection layer is able to reduce index maintenance costs, but requires additional processing and recover-intensive main-memory data structures. With respect to decentralized computing models in modern hardware, it is challenging to maintain an indirection layer. Furthermore, in a key-sorted structure, modifications to an indexed attribute are not treated by the indirection layer and require an index update – resulting in expensive maintenance and random write I/O patterns.

With current indexing techniques, there is no convenient way to simultaneously obtain a beneficial append-only sequential write pattern for index structures and base table data (as listed in Table 2.1) on secondary storage devices.

2.4.1. Version-oblivious Index Equality and Range Search

Indexes in modern multi-version DBMS are version-oblivious by means of missing transaction timestamp (depicted in Figure 2.9). This lack of information results in serious consequences and *can negate effectiveness of additional access paths, due to lowered selectivity and read amplification (RA) for visibility checking*. With regards to Figure 2.10, a small scenario is given.

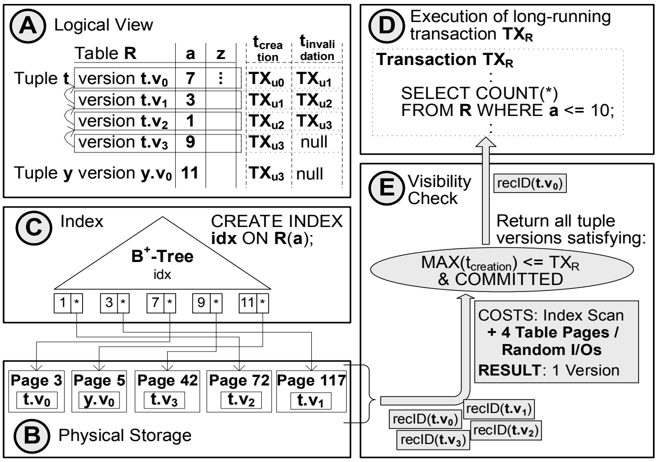


Figure 2.10.: Index search in Multi-Version DBMS and resulting Read Amplification (RA) on Base Tables [RVGP20]

In Figure 2.10-Ⓐ (settled up on example in Figure 2.5), a logical view on a base table R is given. Attribute a of tuple t is modified several times by different transactions (TX_{u0} - TX_{u3}), resulting in physically materialized version records $t.v_0$ - $t.v_3$ with attribute value $7, 3, 1$ and 9. An additional tuple y is inserted for clarification of physically materialized storage. Figure 2.10-Ⓑ depicts an unclustered tuple version storage on random base table pages of a heap structure, since tuple versions are independent record entities. Previously, a key-sorted B$^+$-Tree [BM70] secondary index (outlined in Section 3.1.1) was built upon the attribute a (Figure 2.10-Ⓒ). Its index records reference the tuple version records in the unclustered heap. Figure 2.10-Ⓓ depicts a long-lasting analytical HTAP transaction TX_R, which started before TX_{u1}, TX_{u2} and TX_{u3}. The execution planner and optimizer of the DBMS query evaluation engine decide to use the index from Figure 2.10-Ⓒ. In order to find all tuple version records, which are visible to TX_R, the index is traversed and the data tuple reference is gathered for each matching index record. Referencing tuple version records are fetched from their unclustered physical storage location (Figure 2.10-Ⓑ) and returned to the

MVCC visibility check in Figure 2.10-Ⓔ. *A visibility check forces massive read amplification (RA) in order to gather required tuple version record timestamps for snapshot calculations.*

2.5. Improving Index Search Operation in Multi-Version DBMS

Optimal MVCC designs for base tables (as listed in Table 2.1) efficiently leverage characteristics of secondary storage devices, especially in case of massive data amounts, which are generated by modern workloads, and relatively sparse main-memory. DBMS apply several additional access paths for various purposes. Indexes require frequent maintenance for correctness, however, traditional approaches neglect ideal storage characteristics and result in ineffectiveness. This behavior bases on following indications:

Expensive strictly Key-Sorted Organization. As this index record organization enable fast search operations, its maintenance result in random write I/O patterns and high write amplification (WA).

Reference to Tuple Version Records. Rolling entry points to version chains amplify maintenance costs.

Reduced Selectivity. Downstream visibility checks cascade in high read amplification (RA) as tuple record versions are located along multiple base table pages.

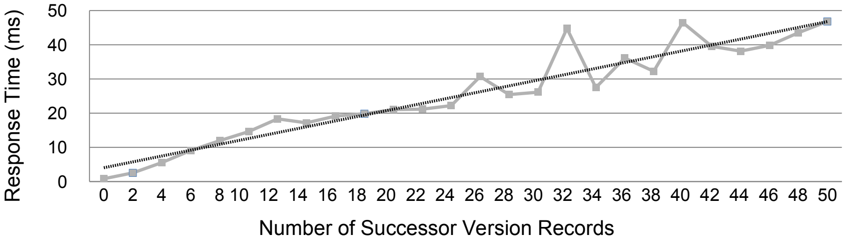


Figure 2.11.: Correlation of Version Chain Length and Response Times.

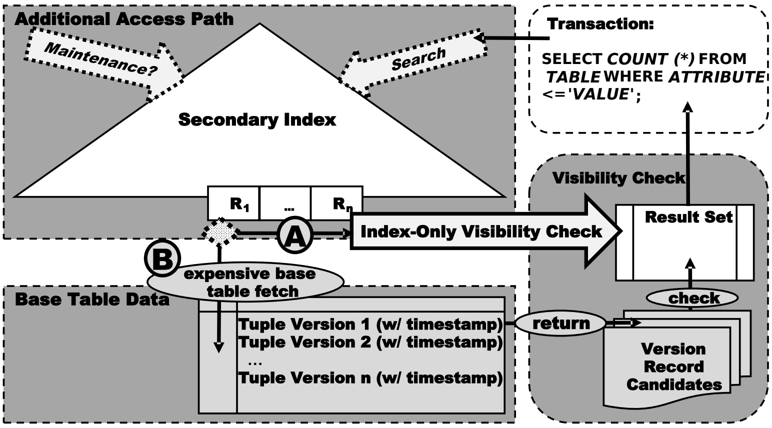

Additional Access Path
Maintenance?
Search
Secondary Index
Transaction:
SELECT COUNT (*) FROM TABLE WHERE ATTRIBUTE <='VALUE';
Visibility Check
R1
...
Rn
A
Index-Only Visibility Check
Result Set
B
expensive base table fetch
Base Table Data
check
Tuple Version 1 (w/ timestamp)
Tuple Version 2 (w/ timestamp)
...
Tuple Version n (w/ timestamp)
return
Version Record Candidates

Figure 2.12.: Index-Only Visibility Checks Ⓐ avoid expensive RA of principally unnecessary base table page fetches Ⓑ.

Additional access paths should allow a key-sorted structure to leverage characteristics of modern storage devices whereby a sequential write pattern with low WA must be enabled. Once an indexing structure facilitates these requirement, it is powerful enough to maintain amplified emergence of tuple version records without a need for layers of indirection, which break optimal MVCC design characteristics or are not suitable for decentralized computing models. Since every tuple version record is physically referenced in the powerful index structure, the only need for processing version chains in base tables is the visibility check – causing massive RA due to sequential processing alongside large tuple record sizes and relative small required timestamp information. Figure 2.11 depicts the linear complexity between version chain length and response time when using a B^+-Tree index in an HTAP scenario. Contrarily, an increased RA in the indexing structure is negligible as required information is space-efficiently packed and asymmetry in storage devices allow fast and parallel read performance. As part of the search operation, a version-aware index structure immediately determines matching tuple versions, which are visible to a transaction snapshot, by a highly selective *index-only visibility check* (as depicted in Figure 2.12).

Chapter 3

Related Work

In Chapter 2, the relevance for high-performance storage and index management structures is exposed. Workload characteristics of modern applications lead to high transactional pressure and raising amounts in data volume to DBMS. MVCC enables high data quality in in HTAP query processing. Furthermore, MVCC designs (as listed in Table 2.1) leverage storage characteristics of Flash-based devices. However, index structures for additional access paths must resist amplified maintenance effort, as it is desirable to index any potentially visible tuple version. In order to obtain a minimal read amplification (RA) on largely sized and randomly spread records in base tables, an ideal index structure cheaply determines the tuple version record, which is visible to a transaction snapshot – i.e. RA on base tables is potentially reduced up to a tuple's version chain length by similar index search effort.

In this chapter, a brief overview of the ubiquitous state-of-the-art B^+-Tree indexing structure is given. Furthermore, current approaches are introduced, which partially tackle required capabilities. In addition, an overview of the designated DBMS and K/V-Store storage structures Snapshot Isolation Append Storage (SIAS) [Got16; GPHB17] and WiredTiger [Mon21] is given.

It is shown, that the structure of Partitioned B-Trees exhibit potentially best properties to fulfill all requirements in indexing massive amounts of tuple version records on modern hardware.

3.1. Storage and auxiliary Index Management Structures

Storage and index management structures allocate information in records, which are located in pages / buffers (compare [LHKN18]). These levels of abstraction unify block-based access on secondary storage media via buffer managers among other reasons. Levels of abstractions are equal for storage as well as index management structures, contained information in the records differ. Whereas storage management structures in this thesis are meant for straight representation of tuple versions in records, index management structures apply an efficient additional access path to desired tuple version records in storage structures. *Hence, major differentiation of both structures is in their application.*

Most key-sorted structures are based on the ubiquitous B$^+$-Tree, which is applied as state-of-the-art indexing structure in many DBMS. Hence, a brief overview is given in the following.

3.1.1. Ubiquitous B$^+$-Tree

B$^+$-Trees are well-known and the most common indexing structure in DBMS. Several structures involve aspects or are fully built upon them, including the applied basic structure in this thesis (Partitioned BTrees [Gra03], compare Section 3.1.4), whereas a brief overview is given. They enable several modification operations on data; i.e. insert, update and delete; as well as efficient data retrieval in equality and range search, due a preserved native sort order in a balanced tree structure with costs, which are based on their logarithmic height by high fan-out index nodes [BM70; Gra11].

B$^+$-Trees represent a balanced tree structure, by means that each traversal operation pass through an equal distance of nodes, i.e. they ensure a robust performance by the maximum height h of a tree. The size of a node can

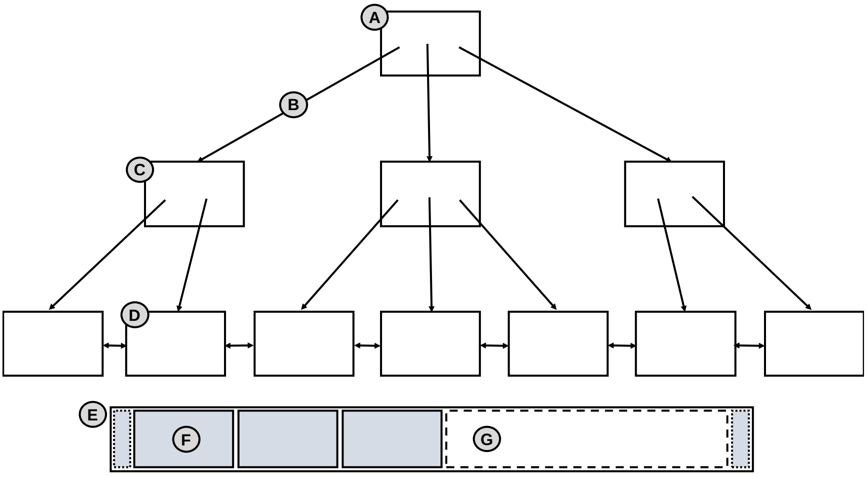


Figure 3.1.: Schematic representation of a B$^+$-Tree (compare [BM70; Gra11]).

vary, generally it is a multiple of the applied transfer and storage size. The structure consists of different types of nodes. With regards to Figure 3.1, *inner nodes* Ⓒ including the *root* Ⓐ apply *reference pointers* Ⓑ in order to indicate several *child nodes* (e.g. Ⓒ is the child of Ⓐ and Ⓓ is the child of Ⓒ). The actual payload is located in *leaf nodes* Ⓓ [1]. Nodes possibly contain sibling pointers, whereas each level in the tree is comparable to a doubly linked list, however, it is not a requirement (e.g. leaves Ⓓ maintain sibling pointers, inner nodes Ⓒ do not). The structure of each node is very similar as depicted in Figure 3.1 Ⓔ. Nodes probably maintain a set of meta data; e.g. size, number of records, reference counters; perhaps sibling pointers (left and right end of the node) and some kind of alpha-numeric sorted *records* Ⓕ. Nodes contain some blank space Ⓖ, which is typically up to half of its capacity. Since records are sorted within a node, binary search enables efficient look-up. Records exhibit *separator key* characteristics in

[1] [BM70] considered the basic B-Tree to locate payload records also in inner nodes, however, nowadays, it is common practice to locate all of them in leaf nodes [Com79]. B-Tree and B$^+$-Tree became synonymous terms.

inner nodes. A child node of a separator key contains only records, with keys similar or greater than itself, but smaller than the next separator key in the node. Leaves contain *tuple* or *index records* of pattern $\{key, value\}$[1], based on their application. A *key* consist of one or a set of attributes (e.g. key $k = \{attr_1, attr_2, \ldots, attr_n\}$), which are alpha-numerically sorted with successive relevance. Therefore, a B$^+$-Tree is able to efficiently search for a set of search key values of a pattern $\{attr_1, attr_2\}$ or $\{attr_1\}$, though not only for $\{attr_2\}$. If its application is data tuple storage, the *value* contains a set of attributes, which are not part of the sort order, otherwise in case of indexing, a value references the tuple record id (or other reference type via indirection layer as outlined in Section 2.4) of the indexed base table tuple record. In order to provide a cost model, following B$^+$-Tree base operations are considered:

Equality Search. The B$^+$-Tree gets traversed from the only entry point (root Ⓐ) to a leaf node Ⓓ. Thereby, the path is followed alongside reference pointers, level by level. Child nodes are selected by binary searching a search key in a parent node. An exact match of an index record key in a leaf node indicates a positive result of the equality search and the record value is returned. There could be quite a few matching equal search key index records.

Range Search. A search interval is defined by a lower and upper key bound. The lower search key is processed as defined in the equality search operation. Index records are processed inside the sorted set in a leaf node and alongside several leaves by following the sibling pointers, while the index record keys match the search interval. Values are processed and returned in the sort order of the tree.

Sorted Insert. An equality search operation is performed with the insert record key. Additional functionality like uniqueness constraints are cheaply

[1]Other record value organization variants, e.g. $\{key, value\text{-}list\}$, are not considered.

examined as part of the equality search. The insert record is placed in the designated sorted leaf node. Thereby, existing index records are relocated to fulfill the sort order. Therefore, the blank space Ⓖ is utilized. Overflowing nodes require additional maintenance – i.e. the node gets split. Half of the records is moved to the new node. The new node is propagated to the parent node. Thereby, it is possible to cause an overflow in the parent node. Modifications require to be protected from side effects, e.g. by applying page locks.

Value Update. Updates to index records, which only affect the value are performed in-place. Therefore, an equality search operation is performed. The value of the equal match index record gets changed. Side effects require to be considered and avoided by locking or indirection layers, as inconsistencies require to be avoided. Updates to search key values result in a costly multiphase deletion of the old record and an insertion of the updated index record.

Record Delete. An equality search operation is performed with the deletion record key. The equal matching index record gets removed from its location in the leaf node. Existing index records become relocated in order to fulfill the sort order. As the blank space Ⓖ exceeds a certain threshold, e.g. half of the node size, nodes underflow and become merged and rebalanced. Correctness of separator keys in parent nodes must be guaranteed. Modifications require to be protected from side effects.

With focus on I/O latencies, costs of considered base operations are derived from following model.

Cost Model. Basic operations in B^+-Trees are subjecting a logarithmic complexity as traversal operation costs depend on their maximum height h_{BT}. Height of the tree depends on the average fan-out F of inner nodes and the number of required leaf nodes L_N, which store a number of index records N. Required nodes are affected by the fill factor f (f_i fill factor of inner nodes, f_l fill factor of leaves), which is the ratio between used and blank space. Hence, it is valid for B$^+$-Trees exceeding capacity of one leaf node, that (compare [Gra11]):

$$h_{BT} = \lceil \log_{F \times f_i} \frac{L_N}{f_l} \rceil + 1 , \quad F \neq 1 , \quad L_N > 1 , \; f_i, f_l \in [0.5; 1] \tag{3.1}$$

A common cost model for equality search in B$^+$-Trees is defined as follows:

$$s_{BT} \approx \log_2 N \tag{3.2}$$

This cost model focus on processing costs, if binary search is assumed. Considering h_{BT} of Equation 3.1, this model is expanded to binary search costs in inner nodes and one leaf node (compare [Gra11]):

$$s_{BT} = \lceil \log_{F \times f_i} \frac{L_N}{f_l} \rceil \times \log_2 (F \times f_i) + \log_2 (\frac{N}{L_N \times f_l}) \tag{3.3}$$

With reference to Equation 2.1, major costs stem from data access on secondary storage devices. Replacing negligible binary search costs $\log_2 \{records\}$ with the I/O costs for reading a node, the cost model is a function of read I/O $R_{I/O}$ and the height h_{BT} of a B$^+$-Tree. Regarding the caching probability for inner nodes p_{ic} and leaves p_{lc}, read I/O can be reduced:

$$s_{I/O_{BT}} \approx R_{I/O} \times (\lceil \log_{F \times f_i} \frac{L_N}{f_l} \rceil \times (1 - p_{ic}) + (1 - p_{lc})) , \quad p_{ic} \gg p_{lc} \tag{3.4}$$

It is more likely for inner nodes to be located in main-memory cache, as buffer managers would prefer more often used nodes. Inner nodes get frequently accessed as part of the traversal operation. Blank space Ⓖ effectively reduces the tree fan-out, due to decreasing fill factor f_i, as well as increases necessary leaf nodes L_N by an average fill factor – and effectively influences the space amplification and search performance, since less data is cached per node and p_{ic} shrinks. However, blank space is beneficial for inserts, since maintenance operations such as node splits are reduced. Any modified node MP_{mod} is asynchronously evicted and written from main-memory to secondary storage device at some point in time. Thereby, an insertion causes at least one write I/O $W_{I/O}$[1], however, due to escalating node splits up to $2{\times}h+1$ are possible, but unlikely.

$$i_{I/O_{BT}} \approx s_{I/O_{BT}} \; + \; W_{I/O} \times MP_{mod} \;, \;\; MP_{mod} \in \{1, 3, \ldots, 2{\times}h+1\} \qquad (3.5)$$

Deletions behave comparable to insertions, as underflowing nodes cause merges. In-place value updates do not cause maintenance operations, thus MP_{mod} is equal to 1. Range search costs are equal to search costs, but require successive reads of leaves alongside sibling pointers.

Assuming massive amounts of data and high rate continuous insertion workloads, it is most likely that only one modification ('-1' in Equation 3.6) is buffered in main-memory until a leaf node is evicted. Vast majority of a node's comprised records ($\frac{N}{L_N \times f_l}$) is unchanged, though gets rewritten. Hence, average write amplification 'WA' of a modified node is:

$$WA \approx \frac{N}{L_N \times f_l} - 1 \qquad (3.6)$$

Suitability to Flash Characteristics. Basic B$^+$-Trees are not capable of leveraging modern storage hardware characteristics. First, in order to keeping up a sorted dataset, B$^+$-Trees perform updates in-place. Randomly modified nodes become asynchronously evicted from buffer management, whereby changes are persisted by rewriting the whole node. On Flash, pages become randomly invalidated and rewritten out-of-place. Second, modifications are manyfold, as reference and sibling pointers, node splits and merges as well as the payload result in altered nodes. Third, marginal modifications require to write one or several whole nodes – resulting in very high WA. Fourth, general average fill factors of approximately 0.67 [RG03] to 0.7 [Gra11] are common, whereby WA is amplified. Last, even if successive logarithmic search effort is very cheap, parallelism is not leveraged.

[1]The probability of multiple modifications per node is ignored, since low ratio in main-memory buffer caches and massive amounts of data makes this case very unlikely – except for serial insertions.

Multi-Version Capabilities. Multi-Version capabilities require transaction timestamps for performing visibility checks. Therefore, a basic B$^+$-Tree needs to maintain creation and invalidation timestamps for each index record – ultimately resulting in amplified in-place updates, increased record sizes and reduced fan-out. Moreover, current and obsolete data is intermingled, wherefore search performance shrinks. A very common practice is to return a set of candidates whereupon visibility is checked on tuple version records in base tables.

3.1.2. Structures with Multi-Version Capabilities

Several searchable structures tackled the capability of multi-version storage and index management [MTT00; ST99]. Applications might maintain different tuple versions in order to perform MVCC or provide constant snapshots for time travel querying. Query predicates and snapshot information constitute different dimensions in search key attribute values, transaction timestamps as well as probably additional application defined temporal dimensions. Well-known representatives are *Multi-Version B-Tree* [BGO+96], *Time-Split B-Tree* [LS90], MV-IDX [GGH+14] or Bi-Temporal Timeline Index [KFM+15], just to name a few. These structures differ in maintenance effort and characteristics, space amplification (SA), searchability of raised dimensions, linkage of related version records as well as general applicability [RVP17a].

Bi-Temporal Timeline Index [KFM+15] enables multi-version capabilities very well in a linearly growing log-based append-only approach of events (activation and invalidation) over transaction time, which are mapped to versions. Checkpoints with visibility mapping information counteract linear growth in effort. Similarly, several application defined temporal dimensions are well covered, whereas its tasks are efficiently met. This approach participates well in column oriented main-memory stores, but there is a lack of cache efficient views on queried attributes in row oriented stores. Moreover, diversity of applied structures and linear reconstruction costs limit its applicability in a general case with complex memory hierarchies.

Multi-Version B-Tree [BGO+96] is an acyclic graph of nodes, comprising tuple versions with validation and invalidation timestamps, hence, one-point invalidation is not feasible. Mutable nodes contain the most recent versions of tuples. Overflowing nodes become immutable but still indexed by search key and temporal dimensions. Its still valid contents are copied in newly created mutable sibling node, which is conditionally split (still overflow) or merged (underflow) in search key dimension. Even if search key as well as temporal dimension are searchable and nodes become immutable, a probably significantly larger part of mutable nodes is randomly forced to secondary storage devices, with the result of undesirable random write I/O, WA and SA.

Time-Split B-Tree [LS90] also enable query in search key and temporal dimensions, however, they form a regular tree structure. One-point invalidation is applied, since successors replace predecessors for unique search key attribute values. Overflowing nodes are either split in search key dimension or in temporal dimension. Based on a timestamp, obsolete version records are moved in – and still valid ones are copied in – an immutable historical sibling node on temporal splits. Version records, which are still valid or get subsequently valid, remain on the former node. Similar to Multi-Version B-Tree, operations yield undesirable random write I/O, WA as well as SA.

MV-IDX [GGH+14] is a regular B^+-Tree extended with in-memory data nodes for every probably visible version record. *Virtual identifier (VID)* in index records reference new-to-old ordered data nodes including timestamps and physical references to base table tuple versions. This logical indirection layer massively reduces maintenance effort on updates, nevertheless insertions and modifications to search key attributes yield random modifications and maintenance of nodes. Despite the significant memory footprint by VID and timestamp, returned candidates are probably excluded from the result set by mismatching query predicates and search key attribute values. Therefore, additional RA on base tables is required. Moreover, undesirable typical SA and WA issues of B$^+$-Tree incur, albeit limited by the indirection layer.

None of the presented multi-version indexing approaches leverage the characteristics of modern hardware in general.

3.1.3. Structures leveraging Modern Secondary Storage Hardware Characteristics

Recent developments in storage and index management structures aim for leveraging modern hardware platforms [KCS+10; LKN13; LLS13; MKM12; Pug90; WPL+18; XCJ+17] as well as secondary storage technologies [Gra04; QCZ+21; WLS21]. Log-Structured Merge Trees (LSM-Trees) [OCGO96] got increased attention, since they are widely adopted as storage management structure in persistent NoSQL K/V-Stores [GD22; Inc22; LC19]. LSM-Trees are composed of several differently sized components (compare Figure 3.2) – typically tiered or in levels arranged separate B$^+$-Tree structures [LC19; SSZA21] with a root as entry point, inner nodes and leaves. Modifications are performed out-of-place in a mutable main-memory mapped component, which is frequently switched and forced to secondary storage devices in a single sequential write and minimized WA.

Increasing number of components need to be traversed on search opera-

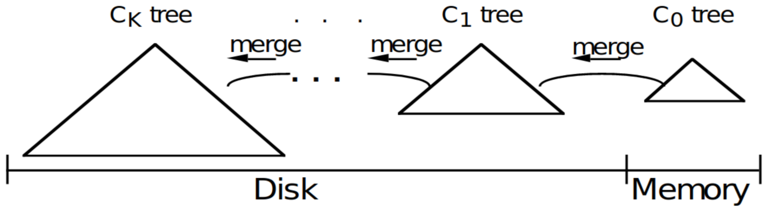


Figure 3.2.: Several $(K+1)$ individual components form a basic LSM-Tree [SR12].

tions. Since components are separate B$^+$-Trees with individually appropriated nodes, LSM-Trees are not able to profit from logarithmic capacity per height properties and common buffering on traversals. Fragmentation is subsequently remedied by merge operations in leveled and/or tiered compaction operations [Cal19; DI18; LC19; SSZA21], yielding frequent and increased WA. Contrary, bLSM [SR12] limit the number of components and utilizes frequent scheduled merging and protects components by bloom filters. REMIX [ZCWJ21] introduces space efficient indexing by key-sorted views on data in multiple fragmented components to deal with growing equality and range search costs. Generally, LSM-Trees need to handle the interplay between RA, WA, SA as well as memory footprint by merge operations to cope with individual workload properties [DAI18a; LC19].

Nevertheless, LSM-Trees are most regarded storage structure in persistent K/V-Stores. Multi-version capabilities are generally applicable to out-of-place replacements of K/V pairs, albeit it is not the focus of LSM. Finally, key uniqueness is natively assumed, although secondary indexing with LSM-Trees is feasible [KKCL17; LC19; ZZC+17].

3.1.4. Applied Basic Structure: Partitioned BTree

Partitioned B-Trees [Gra03] focus on rethinking intrinsic and matured structures in DBMS. The core concept is to introduce an *artificial leading key column* to each index record in a B$^+$-Tree. Inherited design and algorithms allow application of most B$^+$-Tree techniques and optimizations for many purposes – e.g. leveraging modern hardware and workload characteristics.

Its underlying principle is ingeniously simple. Applying an artificial leading key column to index records enables maintenance of partitions within a single B$^+$-Tree by leveraging its intrinsic sort order, as depicted in Figure 3.3. For example, applying the value 4 in the artificial leading key column to any insertion record forces an accompanying search operation in an interval area – i.e. the defined partition – with the value 4 as leading key, whereby partitions with values 0, 1, 2 or 3 remain unaffected. An artificial leading key column is able to store the value of a *partition number*. A partition number can be of any comparable type, which capacity is sufficient to maintain required distinct partitions. [Gra03] suggests an integer of 2 or 4 bytes.

Effectively, maintenance of an artificial leading key column correspond to horizontal partitioning of index records within one single B^+-Tree [Gra03]. Other structures, e.g. LSM-Trees [OCGO96; SR12], enable horizontal partitioning as well, however, they rely on separate B^+-Trees for each partition. Hence, there are major differences in their applicability.

First, separate structures result in schema modifications. Lock-free partition creation is hard to implement this way. Furthermore, execution plans and optimizer require adjustments. On the contrary, an artificial leading key column allows horizontal partitioning as easily as inserting or deleting records [Gra03]. Partition numbers are transparent to other DBMS modules and are dynamically defined by the existence of index records with a specific leading key value. Probably cached bounding partition numbers allow reduced operation effort by atomic operations without any locks.

Second, separate B^+-Trees provide several entry points for traversal, which require to be managed. Since each sub-tree builds a separate tree structure, each component of a LSM-Tree could conversely be considered as a sub-tree of one single tree structure. A very natural approach is to index all components in one B^+-Tree structure with one single root as entry point.

Last, handling all trees as sub-tree in one single B^+-Tree allows commonly used and cached nodes. The capacity is in a logarithmic relation to the height of a tree (compare Equation 3.1). Therefore, space amplification in

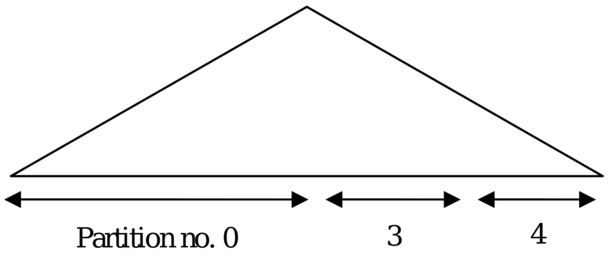


Figure 3.3.: Partitioned BTree – B^+-Tree with partitions [Gra03]

main-memory is reduced by one single tree and cache efficiency is increased (underpinned by example calculation in Appendix A.3).

One might think that maintenance of a partition number would incur additional space and comparison costs. [Gra03] countered this argument by prefix truncation techniques [BU77]. Regularly, just a few bytes on each node is required. Furthermore, truncated prefixes are skipped in binary search and do not increase comparison costs. However, search and sort operations in a Partitioned B-Tree become more complex and expensive, as each partition could contain a desired record. [Gra03] refers that multiple partitions are not the final state of the tree and partition count remains low. In the meantime, search operations require external merge sort and incur multiple traversals.

With regard to Section 2.1, nowadays, secondary storage devices enable high parallelism and exhibit asymmetric read and write latencies. Since merge sort operations allow parallel traversals, an advanced level of data partitioning becomes possible, whereas online merge operations and write amplification (WA) are minimized.

Partitioned B-Trees enable a set of various functionalities. [Gra03] outlines beneficial effort conserving sorting behavior on suspended operations, very fast availability of indexes or cheap bulk load consumption. Furthermore, the author offers an opportunity to consume also smaller insertions into one partition in main-memory and rearrange partitions by online merge operations. Deletions could be handled by some kind of *anti matter*. Applying transaction timestamps to index records would Partitioned B-Trees enable to serve as multi-version store in MVCC. *Partitioned BTrees could provide the key features and structural patterns needed to design a storage and indexing structure for modern workloads and hardware technology characteristics. However, the author does not become concrete in his remarks.*

3.1.5. Prototypical Environments

(Multi-Version) Partitioned BTrees ((MV-)PBT) are practically applicable in almost every B$^+$-Tree based environment. In order to evaluate the full range of storage and index management with MV-PBT, prototypes are implemented in *PostgreSQL 9.0.4 with SIAS* and *WiredTiger 10.0.1*. A short introduction is given.

3.1.5.1. SIAS: Snapshot Isolation Append Storage

Snapshot Isolation Append Storage (SIAS) [Got16; GPHB17] is introduced as an append-only new-to-old ordered storage manager with one-point invalidation and robust performance characteristics on Flash secondary storage technologies.

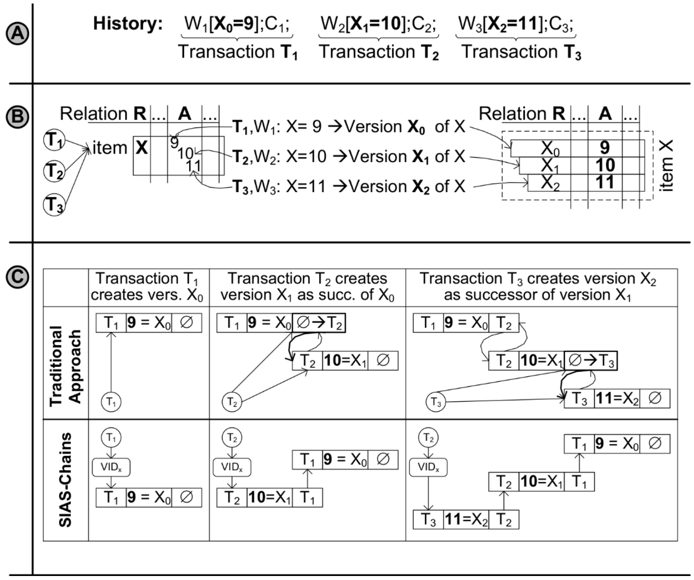

A
History: W1[X0=9];C1; W2[X1=10];C2; W3[X2=11];C3;
Transaction T1
Transaction T2
Transaction T3
B
Relation R ... A ...
item X
T1,W1: X= 9 →Version X0 of X
T2,W2: X=10 →Version X1 of X
T3,W3: X=11 →Version X2 of X
item X
C
Transaction T1 creates vers. X0
Transaction T2 creates version X1 as succ. of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

Figure 3.4.: Snapshot Isolation Append Storage (SIAS) performs updates depicted in Ⓐ by Ⓑ maintenance of tuple versions as Ⓒ out-of-place one-point invalidation (SIAS-Chains) in comparison to traditional two-point invalidation. [Got16; GPHB17]

Contrary to traditional storage management approaches, tuple versions are no longer considered to be individually addressable entities, which require to be particularly invalidated on modification. By means of an unique *virtual identifier (VID)* for each logical data item, related tuple versions form a new-to-old ordered singly linked list. Rolling entry points of logical data items are known by a VID-mapping and each tuple version is aware of its creation transaction timestamp and predecessors physical reference. Considering tuple versions to be a chain of one logical data item enables predecessors to be invalidated by the existence of a visible successor – i.e. one-point invalidation is enabled. By this means, temporally evolving log-based storage management is feasible, which enable minimized WA and a beneficial sequential write pattern to secondary storage devices.

Compared to *PostgreSQL*'s Heap-Only Tuples (HOT) [Pos21], SIAS achieves a robust up to 30% higher transactional throughput, 7 times lower response times, up to 97% write reduction and optimized SA by densely filled pages [GPHB17]. These results are direct effects of applying properties for base table storage outlined in Section 2.3 (Table 2.1).

Index management is proposed to be performed by the logical indirection layer of VIDs and VID-mapping instead of physical references [GGH+14]. This reduces the maintenance effort of indexing the rolling entry points, nevertheless predecessor tuple versions are not immediately obtainable. For instance long-lasting analytical queries in HTAP workloads require to process version chain nodes from its entry point, which lead to successively executed reads and high RA on base table nodes and high latencies.

Based on this prototype, concepts and application of index only visibility checking in DBMS are well verifiable by integrating a prototypical implementation of MV-PBT in the existing B^+-Tree indexing structure.

3.1.5.2. Multi-Core and Memory Optimized B^+-Trees in WiredTiger

WiredTiger (WT) is an open source K/V-Store and the default persistent storage engine in MongoDB [Mon21]. It provides storage and index management by B^+-Trees, former one also for LSM-Trees with *leveled data layout* [Mon21; SSZA21] which build up on components of B^+-Tree. Moreover, schema support in row and column oriented storage with MVCC and SI is supported as well as time travel queries on named snapshots.

A buffer cache is organized in the lock-free B$^+$-Tree structure by hazard pointers, whereof inner nodes reference cached in-memory nodes or disk location. Inner nodes and leaves are organized in flexibly growing pages, which are split and compressed in a reconciliation process on write to secondary storage devices by a block manager in a configurable size. Moreover, this technique facilitates searches and modifications to be performed lock-free by in-memory insert skiplists and update arrays, whereas only fast latches are required on modifications, with the result of beneficial multi-core scaling. B$^+$-Tree is able to exploit its entire potentials in processing and main-memory, whereas pressure on secondary storage devices is massively increased for large updatable datasets. *For these reasons, WT is an appropriate fundament to benchmark bare characteristics of MV-PBT as sole storage management structure in K/V-Stores.*

3.2. Summary and Conclusion

Outlined storage and index management structures vary in their capabilities and features as additional access path in DBMS applying MVCC and SI on modern hardware. As a result, they have shortcomings in multi-version capabilities, complexity in search and maintenance, alignment to modern computing and storage technologies, secondary indexing capabilities or in their specialization for single areas of application. Partitioned B-Trees (PBT) combine search and caching properties of B$^+$-Trees, flexibility in the application of partition management, for instance to perform out-of-place updates by mentioned *anti matter* [Gra03]. Certain proximity to LSM-Trees, as well as their fields of application, and the idea of PBT serving as *multi-version store* [Gra03] facilitate the application as storage and index management structure. Even though the conceivable proximity to LSM-Trees, PBT leverages characteristics of a single tree structure rather than fragmenting related data in several individually managed structures (compare calculations from Section A.3). In the main contributing Chapter 4, characteristics of PBT are leveraged to enable multi-version indexing on modern storage technologies.

Chapter 4

Multi-Version Partitioned BTree

In this main chapter, the structure of Multi-Version Partitioned BTree (MV-PBT) is outlined. MV-PBT – an enhanced version-aware index and storage management structure based on the structuring pattern of Partitioned BTrees [Gra03] – exposes near optimal key-sorted structural features for modern workload and hardware characteristics. Whilst the author of [Gra03] argues about individual areas of application for Partitioned BTrees, a holistic design proposal, prototypical implementation and evaluation of potentially related and raised concepts are not provided.

On this occasion, a holistic storage and index management structure is concretely designed, following the claims of preceding chapters. Horizontal partitioning is leveraged for version-aware and write-optimized indexing (**RQ1**) as well as storage management (**RQ2**). Moreover, MV-PBT requires elementary considerations about steady performance characteristics and maintenance raised by **RQ3**, **RQ4** and **RQ5**.

4.1. Multi-Version Indexing Design Concepts

Auxiliary index structures offer an additional access path to queried data, which is located in base tables. Maintenance of tuple version records in base tables entail additional complexity and costs in maintenance or selectivity in data access by index management structures (as outlined in Sections 2.4 to 2.5). In order to address **RQ1**, Multi-Version Partitioned BTree (MV-PBT, Section 4.3) is designed as version-aware storage and index management structure, which leverages characteristics of modern hardware technologies.

First, version-aware index management structures appreciate tuple versions as individually accessible entities and optimize selectivity by *index-only visibility checking* (Section 4.3.6). Therefore, index records require extended information, like a record's *transaction timestamp*, *tuple version ordering* and *performed operation* besides well-known *search key attributes* and *logical* or *physical references* to tuple versions in base table as *value*. Second, this extended information requires to be maintained by leveraging *characteristics of modern hardware technologies* (introduced in Section 2.1) for a broad range of applicability in *modern workloads* (introduced in Section 2.2).

MV-PBT leverages the structuring pattern of *Partitioned BTree*s [Gra03] (compare Section 3.1.4) to maintain *out-of-place* index version records in an appropriate succession of preferred *version ordering* (Section 2.3.1.2) by adaption of their *partition number*s (in *partitioned keys* as outlined in Section 4.2.1.2). Index version records are annotated with *transaction timestamps* of the creating transaction, whereby a *transaction snapshot*'s related version record is identified, if following the logically maintained *new-to-old version ordering* (Section 4.3.3) of tuple versions by manipulating the index record's

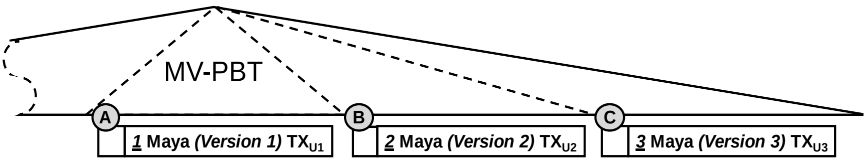


Figure 4.1.: Concept of Multi-Version Partitioned BTree (MV-PBT).

partition number. Beneficial append-only *one-point invalidations* (Section 2.3.1.3) are performed by applying some '*anti matter*' [Gra03] to specific *index record types* (Sections 4.2.1.3 and 4.3.1) for different operations (Section 4.2.3). In doing so, an index version record located in partition number 2 (compare Figure 4.1 Ⓑ) is invalidated and replaced by a more recent record in partition number 3 (Ⓒ), if reads are performed in reversed succession (from 3 to 0 result in ordering Ⓒ, Ⓑ and Ⓐ). Detailed ordering conventions are outlined in Section 4.3.4.

Maintenance of out-of-place index version records accompanies well with characteristics of Flash secondary storage devices (Section 2.1.2). Structuring pattern of *Partitioned BTrees* [Gra03] assumes responsibility for several purposes as introduced in Section 3.1.4. MV-PBT appropriates this capability on its own fashion. Whilst new index version records are maintained in new partitions, elder ones natively become immutable. Updatable data is preferably kept in fast main-memory, whereas immutable data is evicted to secondary storage devices. Native key-sorted order enables a sequential write pattern of immutable data in leaf nodes by singularly forcing partition related nodes to secondary storage devices (Section 4.2.2). For instance, while index version records are inserted with partition number 3 in Figure 4.1 in main-memory, preceding partitions 0, 1 and 2 are located on Flash secondary storage devices.

Concepts of [Gra03] are combined in a consistent version-aware storage and index management structure for modern transaction processing in DBMS, which apply MVCC and SI, and considering modern hardware characteristics in Multi-Version Partitioned BTree (MV-PBT). With regards to the hardware characteristic-leveraging component of **RQ1**, first, a concrete proposal for write-optimization with the concepts of *Partitioned BTrees* is given in Section 4.2. Leveraging horizontal partitioning for write-optimization on modern Flash-based secondary storage devices significantly differs from proposed applications in [Gra03] and constitutes an innovative application of the matured structural properties of *Partitioned BTrees,* denoted as PBT.

Structural properties of the proposed write-optimized instance of PBT are leveraged to introduce an index-only visibility check (second part of **RQ1**), whereas selectivity of result sets are optimized in MV-PBT. Moreover, MV-PBT is considered as multi-version storage management structure (**RQ2**), since it inherently addresses storage properties and visibility checking. Partitioning in MV-PBT requires elementary considerations about steady performance characteristics and maintenance, raised by **RQ3**, **RQ4** and **RQ5**.

4.2. Write-Optimization with Partitioned BTrees

Alpha-numeric key-sorted data structures, like B^+-Trees, are very common in DBMS and K/V-Stores, since they enable fast equality as well as range search in an modifiable dataset. However, in order to sustain alpha-numeric sort order, B^+-Trees require special considerations and additional work. Consequent downsides of this structure emerge in persistent layers of the memory hierarchy. Writing one node to secondary storage devices result in high write amplification (WA), due to following essential facts:

- Modern workload characteristics lead to a low share of buffered data in main-memory compared to massive amounts of data on secondary storage media of conventional data nodes.
- Version records amplify index maintenance, due to MVCC storage characteristics in Table 2.1.
- Mandatory blank space in inner nodes and leaves increase WA and SA.
- Modification and maintenance operations result in massive WA, since most written data was already persistent.
- Random modifications lead to invalidation and write operations on incidental pages on Flash secondary storage devices and effectively increase latencies and internal WA (Section 2.1).

Based on ideas in Section 4.1, appropriate partition management with the structuring pattern of Partitioned BTrees [Gra03] could solve these problems by an innovative application of write-optimization with PBT. Estimated characteristics are formulated in following hypotheses:

Hypothesis 1 (H1)
PBT is able effectively reduce WA, since each modification is written once.

Hypothesis 2 (H2)
PBT achieves a beneficial sequential write pattern to secondary storage devices.

Hypothesis 3 (H3)
PBT minimizes blank space on leaf nodes and ultimately avoid massive SA.

In order to prove **(H1)**, **(H2)** and **(H3)**, PBT is introduced as write-optimized version-oblivious index and storage management structure for modern hardware technologies.

4.2.1. Data Structure and Components

According to [Gra03] (outlined in Section 3.1.4), PBT also maintains partitions within a regular B$^+$-Tree – i.e. its underlying structure is equal to the characteristics outlined in Section 3.1.1, however partitioning within one tree structure yield new opportunities and enables a directed determination of structural effects.

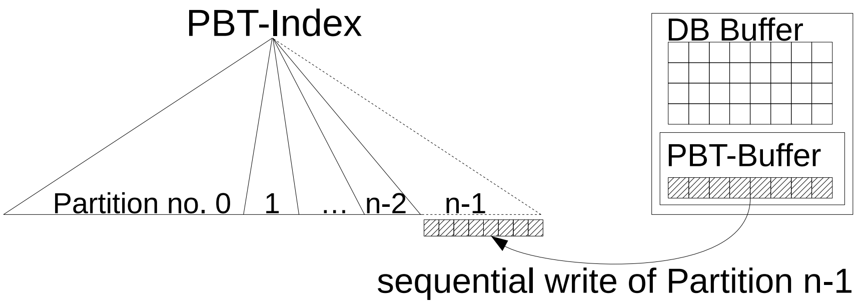


Figure 4.2.: Write-optimization with PBT – Updates are buffered in main-memory PBT-Buffer and sequentially written to secondary storage devices. [RVP17b; RVP19]

Regular B$^+$-Trees random write I/O pattern and WA are direct effects of its strict preservation of alpha-numeric sort order on updatable data in a tree structure that is much larger than available main-memory. With PBT (depicted in Figure 4.2), every modification is directed to a partition, that is small enough to be located in main-memory. A designated area in the database *Buffer Manager*, i.e. *PBT-Buffer*, collects updates in leaf nodes.

Whilst introduced properties of PBT are in accordance with transaction processing purposes in [Gra03], write-optimization by sequential writes is only considered in PBT. Once a certain threshold is reached, the main-memory partition becomes immutable to further modification and gets sequentially written. However, continuous modifications are directed to an atomically switched succeeding partition – leading to a native and transparent horizontal partitioning behavior.

In the following, essential structural components and properties of PBT are outlined in detail.

4.2.1.1. PBT Cached Meta Structures

According to [Gra03], PBT's auxiliary meta information is entirely contained in the B$^+$-Tree structure. For instance, the most recent partition number of a PBT could be identified by searching the rightmost index record in the tree structure [Gra03]. Since this information is frequently required and its memory footprint is very low, auxiliary meta data structures are

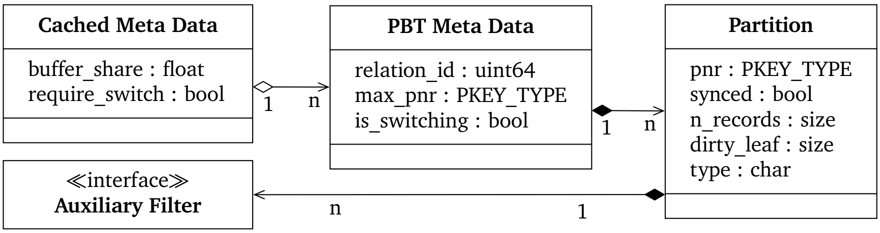


Figure 4.3.: Auxiliary recoverable PBT cached meta data structures.

cached in main-memory (inspired by '*integrity constraints*' in [Gra03], an excerpt of a specific proposal for PBT is depicted in Figure 4.3). Generally, PBT cached meta data structures require neither locking for any atomic operation nor additional logging of modifications, due to its straightforward and fully recoverable design. Contrary, LSM-Trees require disadvantageous lock-assuming schema modifications for horizontal partitioning and log-based storage. The applied approach in PBT is completely transparent to further DBMS modules, since the horizontal partitioning is anchored in a B$^+$-Tree structure.

A brief introduction in the cached meta structures is given (depicted in Figure 4.3) and also repeatedly referenced in the following sections. In a DBMS, multiple PBTs (which possess *PBT Meta Data*) are maintained and commonly share *Cached Meta Data* and a fraction of buffer frames, which are frequently synchronized on a partition switch, as outlined in Sections 4.2.1.4 and 4.2.2. Each PBT composes of *Partition*s, whereas the most recent partition number (*max_pnr*) is manyfold appropriated. Partitions comprise *Auxiliary Filter* structures for data skipping methods, as outlined in Section 4.4.3.

4.2.1.2. Artificial Leading Key Column and Partitioned Key Type

B$^+$-Tree structures inherently maintain an alpha-numerical sort order. [Gra03] introduces *partition numbers* in *artificial leading key columns* of records in order to natively maintain horizontal partitions. Profound changes in search key attribute values of every contained record require comprehensive considerations in a specific implementation, like PBT, as outlined in the following.

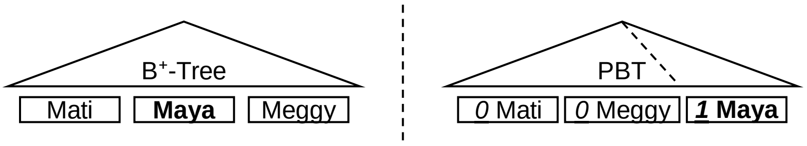


Figure 4.4.: Horizontal Partitioning within a single B$^+$-Tree.

Partitions are principally defined by the existence of index records with a specific artificial leading key value. Hence, partitions are natively maintained by manipulation of the index record's leading key column with the result of horizontal partitioning in one single B^+-Tree (as depicted in Figure 4.4). Since multi column index records in a B^+-Tree structure are generally ascending or descending ordered column by column, [Gra03] suggests to prepend an artificial leading key column to each index record, which is filled with the *partition number*.

Adding columns equate to unintentional schema modifications, however, horizontal partitioning in PBT is a desired structural behavior. An additional column adversely affects record layout descriptors and subordinated characteristics of frequently performed algorithms, e.g. comparison and compression.

Composite *partitioned key* attribute types are more favorable. In a PBT, the first attribute is extended by a fixed size partition number (as depicted in Figure 4.5). There are several possible implementation options. If an attribute value is copied into a partitioned search key, it is possible to relocate the value by an appropriate number of bytes – i.e. by the size of the partition number type. This behavior is enabled by defining structures (similar to Listing 4.1). Since the number of data types in a DBMS is manageable, this is a sound solution. However, handling a reference type this way (as depicted in Listing 4.1), this would cause pointer chasing and performance issues, as comparisons are frequently performed. Therefore, the partition number should be located at the beginning of the referenced area, whereas cache

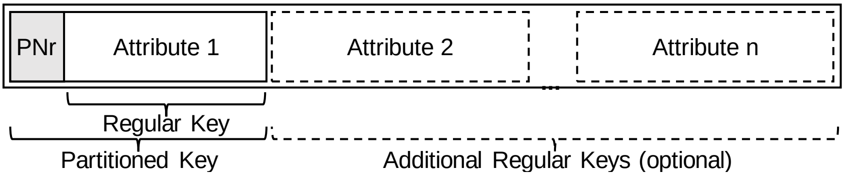


Figure 4.5.: PBTs convert the first attribute to a Partitioned Key – consisting of a partition number (PNr) and the regular key.

```
#define PKEY_TYPE uint16_t
typedef struct PKEY_4byte {
  PKEY_TYPE   partition_number;
  uint32_t    value;
} PKEY_4byte_t;
typedef struct PKEY_byRef {
  void  *value; // ptr to PKEY'd reference
  size_t size;  // increment by sizeof(PKEY_TYPE)
} PKEY_byRef_t;
```

Listing 4.1: Sample Partitioned Key

efficient comparisons as well as cheap raw storage and prefix truncation is enabled.

Comparisons are frequently performed operations, e.g. if binary searching an index node as part of an equality search. Search keys contain a set of attribute value and operator qualifiers, which are successively compared to index record key attributes. Based on record layout descriptors, raw keys on index nodes require to be reconstructed. Partitioned key types simply overlay the first attribute descriptor for record reconstruction in PBT. Since data types do not follow equal comparison strategies, it is very common to apply a strategy programming pattern (compare [GHJV95]) with comparator functions. Partition numbers of partitioned keys require to be separately compared from its regular key, as its strategy might differ. However, in contrast to separate artificial leading key columns, successive comparison effort is reduced, since the number of attributes remain equal and partition number and the first attribute are co-located in cache (compare processing costs in Figure 4.6).

Partition numbers require to be transparent to further DBMS modules, hence the partition number requires to be hidden from a partitioned key attribute. Since the regular key is entirely contained in the partitioned key, its attribute value is cheaply returned as an offset without additional processing costs.

Additional storage costs of partitioned keys depend on several factors. Beside fan-out and fill factor, the partition number type (PKEY_TYPE in

Listing 4.1 and Figure 4.3) is considered to be a factor, as it increases the record size and consequently shrinks the tree fan-out. An appropriate type seems to be an 2-byte unsigned short integer with a maximum of 65535 partitions, since it is a good trade-off between storage costs and flexibility in partitioning.

In a regular B$^+$-Tree structure, this might have following effects[1]. In a read optimized B$^+$-Tree, its node fan-out shrinks about 9%. However, whilst its height is not affected, traversal costs remain equal but scan costs slightly increase as cache efficiency shrinks. Considering continuous insertions, a read optimized B$^+$-Tree layout is inadequate, since modifications cause node split operations, whereas the average fill factor per node shrinks by up to 50% in a regular B$^+$-Tree. Contrary, partitioning enables desirable append only behavior, whereby a read optimized layout is reconditioned within each partition. As a result, more records are treated per node and the tree fan-out is leveraged by a nearly optimal fill factor. Since the PBT structure is equal to B$^+$-Trees, techniques like prefix and suffix truncation [BU77; Gra11] become applicable. Separator keys in inner nodes only purpose is to *'separate'* sub-trees by discriminating keys. Suffix truncation allows to

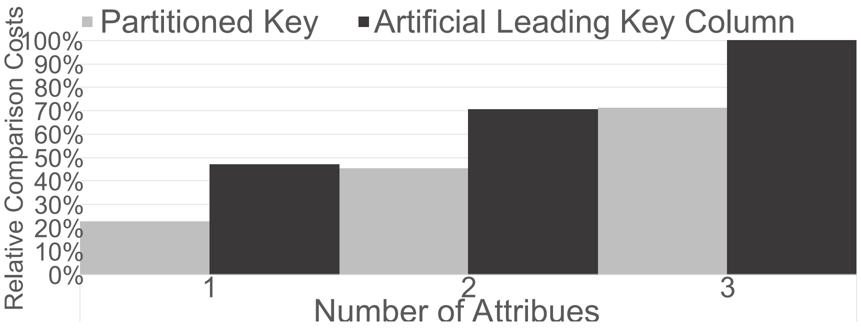


Figure 4.6.: Relative Comparison Costs for both approaches. Partitioned Keys save an iteration and leverage cache efficient operations.

[1]Assuming *4 kB* node size, *4 byte* keys, *16 byte* reference pointers and *2 byte* partition numbers in this calculations. Meta data is not considered.

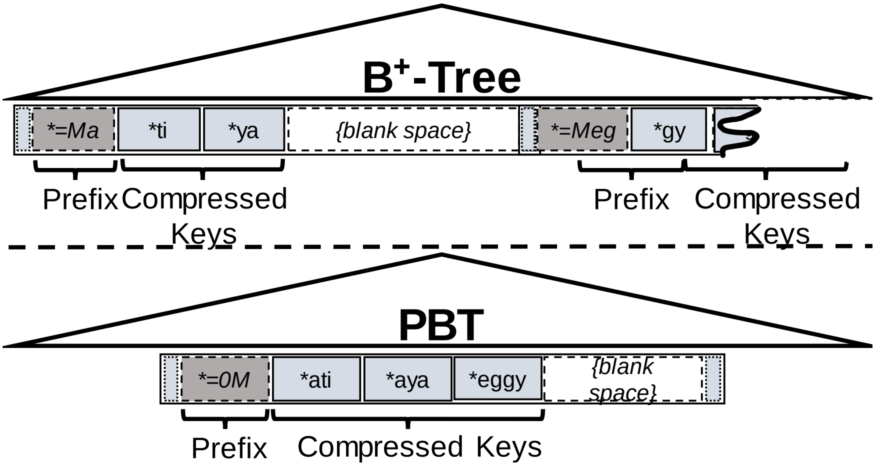


Figure 4.7.: Compression techniques become more valuable in PBT, since partition numbers are part of common prefixes.

store just enough leading bytes of a key to separate sub-trees on traversal. In PBT, these discriminating bytes could be partition numbers, since several record keys begin with an equal prefix – at least in the root and some inner nodes. The fan-out is massively increased, especially in case of large keys. Prefix truncation contrary exhibit reduced storage cost in leaf nodes, since partition numbers and perhaps succeeding common bytes of all record keys on a node are practically always truncated and stored once. The alpha-numeric sort order of a dataset within a node and inside the whole tree intensify this truncation effect, as similar keys are co-located. Furthermore, truncation effects become more valuable based on the enlarged common prefix of the partition number (compare Figure 4.7, whereas B$^+$-Tree relies on node split for adequate prefixes). *Considering the outlined truncation and fill factor effects, additional storage costs are negligible, since index node capacities are aligned to multiples of block-addressable storage sizes. Possible effects are evaluated and depicted in Section 4.2.4 Figure 4.15.*

4.2.1.3. Index Record Types

PBT is an append-based storage and index management structure, which is capable to handle high-rate continuous insertion workloads. The potential is obtained through strict maintenance rules and immutable persistent partitions. However, in a tuple life cycle, modifications are an elemental feature in such structures. Besides search operations, tuples are created, updated or deleted. In a regular B$^+$-Tree with strict alpha-numeric sort order, modifying operations are performed at a designated location, as outlined in Section 3.1.1. In combination with such behavior, Hypotheses **(H1)**, **(H2)** and **(H3)** are not attained.

[Gra03] introduced the term *'anti matter'* as some special kind of deletion markers, which are inserted and processed similar to regular records, but with contrary meaning. Deletion markers (alias tombstones) are familiar from append-based structures, such as LSM-Trees or SIAS (compare [Got16; OCGO96]. Since version-oblivious LSM-Trees are considered to simply replace data with more recent unique record key identifier, special invalidation values are sufficient. Subsequent search operations treat these equal key records as invalidated until equal key records get finally deleted by merge operations at an appropriate point in time [OCGO96; SPSA20]. Contrary, a special type of tombstone records in SIAS define invalidation of all predecessors in a version chain with equal virtual tuple identifier (VID) [Got16].

PBT is designed as a full featuring storage and index management structure. Whereas simple invalidation is sufficient for unique identifiers in data storage, native non-uniqueness and multi-attribute treatment is necessary for indexing structures. Hence, PBT introduces several record types – featuring the whole range of operations in a tuple life cycle (compare Figure 4.8):

Regular Records are inserted on creation of a new tuple. Hence, there must not be any preceding record in a tuple life cycle. Its behavior is comparable to an inserted record in an regular B$^+$-Tree. It consists of a *partition number*, one or several *key attribute values* and a *record value*. The latter might refer

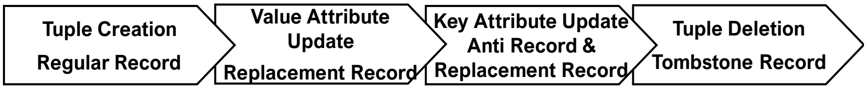


Figure 4.8.: Various operations over tuple life cycle result in practicable insertion of different index record types.

to a data tuple by logical or physical record identifier or contains the value data itself – based on the application.

Replacement Records are inserted on modification of a *value attribute*, which is not part of the record key, i.e. there is a preceding record that gets replaced by a new value. In principle, its record layout is comparable to other record types with a *partition number*, one or several *key attribute values* but up to two *record values*, because its relation to a specific predecessor must be clarified. Generally, one record value contains the new data associated with the tuple and the other contains the association value of its preceding record. Therefore, if the record key and the latter value are equal to its predecessor, its association is ensured. However, an association is also ensured if, (a) PBT is applied as storage management structure, (b) a primary key or uniqueness constraint is defined, (c) logical reference pointer are applied or (d) predecessors must be invalidated by Anti Records. In order to save storage cost and reduce SA, the second (old) record value is probably hidden in such situations.

Anti Records are pure anti matter and inserted on modification of key attribute values in combination with Replacement Records. Since the key attribute values changed, a Replacement Record is not sufficient to invalidate its predecessor. Therefore, an Anti Record is inserted. Its record layout consists of a *partition number*, one or several *key attribute values* and one *record value*, which is equal to its predecessor record value. In contrast to other append-based structures, thus PBT is able to feature whole tuple life cycle, even on key modifications and is predestinated as index management structure. Nevertheless, Anti Records require special treatment in version-oblivious PBT in combination with MVCC-SI base table design in Table 2.1.

Tombstone Records are pure anti matter and inserted on deletion of a tuple. There is high correlation with the utilization of Anti Records, however, there is a massive logical difference. Tombstone Records mark the end of a tuple life cycle, i.e. no successor of a tuple is possible. Tombstone Records consist of a *partition number*, one or several *key attribute values* and a *record value*, which subjects equal constraints like association record values of Replacement Records. Likewise, Tombstone Records require special treatment in version-oblivious PBT in combination with MVCC-SI base table design in Table 2.1.

Additional storage and processing costs require to be considered. First, while LSM-Trees typically insert special deletion marker values to describe an invalidation of records with a specific key, PBT aims to distinguish different operations by record types, by means of enabling full append-based indexing features. Ordinary B$^+$-Tree records maintain meta data, which contain several property information as bit flags. Usually, some padding bits are not in use or special properties are not required in PBT, e.g. deletion flags. In order to represent all record types in PBT, 2 bits are sufficient. The first bit could indicate, if preceding record values are replaced by a new value, a second bit denotes pure anti matter. In this manner, Regular Records are denoted as ’00’, Replacement Records as ’10’, Anti Records as ’11’ and Tombstone Records as ’01’. Record type flags in combination with further property information of the tree derive consequent behavior of PBT, e.g. the number of required record value fields in Replacement Records for compression. Maintenance of record types is enabled by very cheap bitwise operations and usually no additional storage costs.

Second, append-based value replacements yield additional storage costs. However, this is not its final state. Temporally bounded persistent secondary storage media is sufficiently available, due to cheap acquisition and operating costs. Furthermore, split policies in regular B$^+$-Trees yield SA up to twice of its actual dataset. Append-based storage enables near optimal fill factor on nodes as well as WA. Moreover, replacements encode additional information, i.e. the modified value, which can be leveraged for further purposes, like versioning in MVCC. Anyways, a read optimized arrangement with near optimal SA can be restored at an appropriate point in time.

Third, record values in Replacement, Anti and Tombstone Records additionally increase SA, since they guarantee tuple association even in non-unique index management. Generally, it is an intermediate circumstance. Equal behavior in other append-based storage structures is not natively enabled for non-uniqe index management, since search key uniqueness is assumed, which could be achieved by additional virtual identifiers. PBT natively supports non-uniqueness in index management. Tuple association is probably enabled by further constraints and additional value fields are not required at all. Furthermore, association record values can be stored as delta or in compressed shape. Anyways, with equal feature support like LSM-Trees, PBT record types take up comparable space.

Last, equality and range search operations require additional processing. Obviously, increased number of records yield increased processing costs. Since I/O costs are considered orders of magnitude higher than computing costs (Equation 2.1), logarithmic I/O costs of B$^+$-Tree search algorithm take effect. Searches in partitions are perfectly parallelizable in merge sorted operations and leverage multi core processing characteristics as well as asymmetry and high internal parallelism in modern Flash storage (compare Section 2.1). Furthermore, native partitioning order and data skipping methods reduce overhead in additional processing costs of maintained records. Moreover, caching effects of partitioning speed up modifying operations by orders of magnitude, whereas resources become available for search operations and overall throughput is increased.

Additional costs of record types for different operations in a PBT are manageable, since they enable modifications in beneficial append-based storage and index management.

4.2.1.4. PBT-Buffer Management

Out-of-place modifications to the dataset by different record types (Section 4.2.1.3) in PBT incur increasing space requirements. Limited main-memory capacity is organized by a buffer manager and different replacement policies as introduced in Section 2.3. However, regular buffer managers and replacement policies are not aware of PBT partitions. Traditional incidental approaches are incompatible with intended write-optimization in PBT.

PBT rules out a very hot subset of buffer frames from regular replacement policy. In the *PBT-Buffer*, leaf nodes of most recent mutable partitions of all PBTs in a DBMS are commonly cached. Inner nodes and immutable leaf nodes remain under control of the regular replacement policy. This design decision has following effects (compare Figure 4.9):

Root and upper levels of inner nodes Ⓐ are almost never modified, but frequently required, since search operations in B^+-Tree structures traverse the tree from its entry point (the root node) via inner nodes towards the leaf nodes. Assuming a fan-out of 100, a tree height of 4 and 1 million randomly distributed equality searches, the probabilistic number of requests on a node

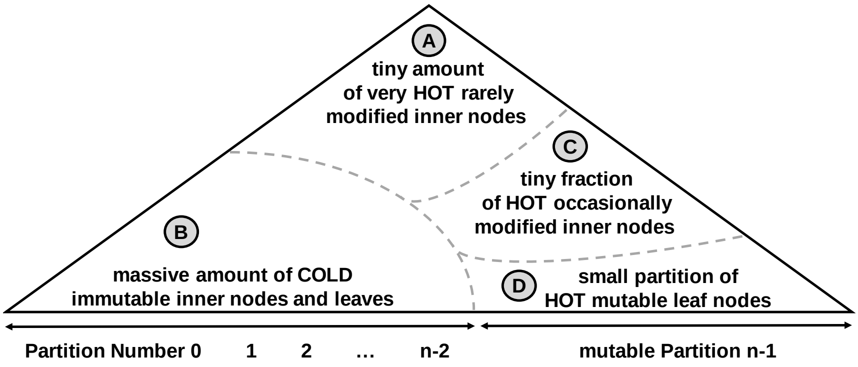

A
tiny amount of very HOT rarely modified inner nodes
C
tiny fraction of HOT occasionally modified inner nodes
B
massive amount of COLD immutable inner nodes and leaves
D
small partition of HOT mutable leaf nodes
Partition Number 0 1 2 … n-2
mutable Partition n-1

Figure 4.9.: PBT-Buffer's replacement policy yield hot/cold separation. [RP22]

per level is 1 million on the root, 10 thousand on each inner node in the second level, 100 on each inner node in the third level and only one for each leaf. Regular replacement policies honor this behavior by a high hit rate on upper tree levels, as their usage count is very high and their memory footprint is comparably low. In this case, therefore, half of the traversed path is cached on each equality search.

Lower level inner nodes and leaves of immutable partitions Ⓑ are never modified by basic operations of a PBT, hence their contained data is already persisted and it is not necessary to write their contents on replacement. Moreover, it is very unlikely for these nodes to get immediately reused. Since Flash based storage devices close the access gap in memory hierarchy, buffering is not valuable. A regular replacement policy select them as victim buffer frames for eviction.

Inner nodes guiding to mutable partition leaves Ⓒ are frequently visited and get occasionally modified, due to update intensity of referenced leaf nodes. High usage count practically prevent them from frequent eviction, however sporadic flushes might occur. Since these writes are limited to a focused area of the tree (compare Section 2.1 *Advantages in sequential write I/O*), random write I/O is bearable. Alternatively, these inner nodes could be located in the PBT-Buffer, though a special treatment of commonly used inner nodes by several partitions would require additional complexity.

Leaf nodes of mutable partitions Ⓓ are treated by the PBT-Buffer replacement policy, which is a set of buffer frames that are not under control of the regular replacement policy. This area is shared for every PBT in a DBMS, hence partition sizes are self balancing. The PBT-Buffer prohibits eviction of the most recent mutable partition of a PBT, whereas its leaf nodes remain in main-memory buffer frames. Once a partition becomes immutable by an partition switch, it is possible to evict its defragmented and cleaned leaf nodes.

Modifying as well as read only operations benefit from excellent buffering of a whole traversal path to the most recent partition (alongside sections Ⓐ, Ⓒ and Ⓓ in Figure 4.9). Furthermore, PBT-Buffer replacement policy effectively saves mutable leaf nodes in buffer frames from early eviction of intermediate page states, whereas **(H1)** is facilitated. However, eviction of leaf nodes in a PBT-Buffer require special treatment on partition switch in order to guarantee **(H2)** and **(H3)**. This process is outlined in Section 4.2.2.

4.2.2. Partition Switch & Sequential Write

Write-optimization with PBT is achieved by strict insertion of special index record types (Section 4.2.1.3) in a designated mutable main-memory allocated partition on any modification. Since the main-memory share of the PBT-Buffer frames is a finite factor in continuous modifying workloads, partition leaf nodes getting frequently flushed and evicted to secondary storage devices – i.e. mutable main-memory and persistent representations are getting synchronized in order to reclaim space in the PBT-Buffer.

Objective of this operation is to efficiently write an information delta to secondary storage devices whilst ensuring the aims in hypotheses **(H1)**, **(H2)** and **(H3)**. Furthermore, an elementary requirement is to guarantee continuing concurrent transaction processing with marginal interference.

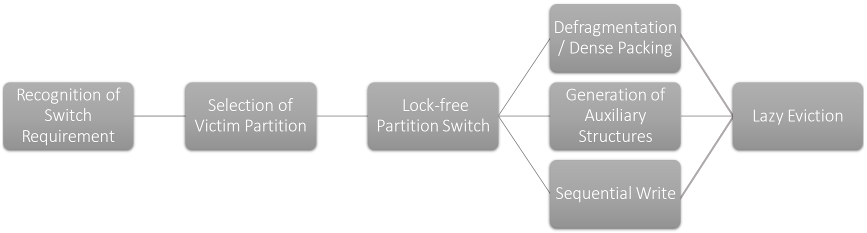

Recognition of Switch Requirement
Selection of Victim Partition
Lock-free Partition Switch
Defragmentation / Dense Packing
Generation of Auxiliary Structures
Sequential Write
Lazy Eviction

Figure 4.10.: Partition Switching process is asynchronously performed by a parallelizable background worker with marginal influence on payload.

PBT facilitates the requirements by atomically switching a partition number and sequentially writing a read optimized serialization of leaf nodes. The entire process is asynchronously performed in multiple steps whenever a certain threshold of dirty buffers in the PBT-Buffer is reached:

- *Recognition of switch requirement*
- *Selection of a victim partition*
- *Lock-free Partition Switch*
- *Defragmentation and Dense-Packing*
- *Generating auxiliary structures as a natural by-product*
- *Beneficial sequential write of leaf nodes*
- *Lazy Eviction and flexible assignment of PBT-Buffer Frames*

The whole process facilitates asynchronous execution by a parallelizable background worker (as depicted in Figure 4.10) and lock-free operations. In the following, challenges and contributions of each step are outlined in detail.

4.2.2.1. Switch Requirement Recognition

Recognition of switch requirement is a crucial prerequisite and its appropriate conduct is a workload dependent factor. A lightweight approach is to define a configurable threshold of dirty leaf nodes in the PBT-Buffer. Exceedance of the threshold as a result of a modifying transaction triggers an ideally asynchronously performed switching process in a background worker – i.e. payload proceeds with marginal interference whilst the PBT-Buffer provides enough space in main-memory.

4.2.2.2. Victim Partition Selection

Selection of a valuable victim partition likewise is a workload dependent factor. Since all PBTs in a DBMS are under control of the PBT-Buffer replacement policy, they commonly share the main-memory buffer frames.

While this property is very beneficial in administrative effort[1], partition selection requires to fulfill several aspects and provide an option for different replacement strategies. Partitions strive for maximum size allocation in main-memory, however, it is necessary to allow a parity growth as well as maximum space reclamation in high data injection periods. Adaptive approaches are still an open research area, however, an initial approach is a round robin method of valuable partitions, beginning with the most space occupying partition.

4.2.2.3. Lock-free Partition Switch

Lock-free partition switching is enabled, since the most recent partition number is well-known in a PBT (as outlined in Section 4.2.1.1). In contrast to LSM-Trees, this is a major benefit as component switches are typically lock-assuming schema modifications. PBT atomically increments the most recent partition number in main-memory meta structures without precautions, since it could be simply recovered in case of failure. The required B^+-Tree structure is already existent and operations are logged anyways. Modifications of downstream processes are treated by the newly generated main-memory partition (as depicted in Figure 4.11a Ⓓ). Generally, concurrent transactions can proceed without any atypical interferences, with one exception: modifying transactions could intend to insert an index record of any type into an already defragmented and perhaps already cleaned leaf node. Moreover, concurrent modifications could delay and interfere with the background worker process.

A very straightforward approach to avoid interferences is to prohibit user initiated modifications of the victim partition whenever the most recent partition number has been switched. Since interferences could break conditions of **(H1)**, **(H2)** and / or **(H3)**, such an operation would override its partition number in the search key to the new most recent one and re-traverse the

[1]Comparing PBT with LSM-Trees, both structures facilitate to keep a fraction of data in main-memory. In this context, whilst LSM component sizes require a fixed configured threshold per tree structure, PBT allows a very straightforward self balancing shared approach.

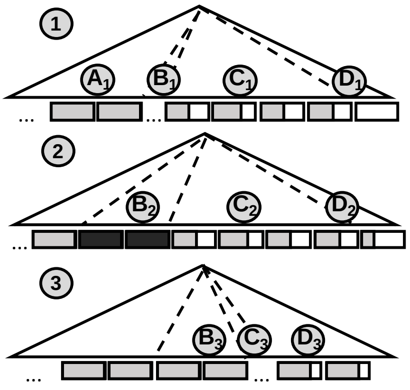
PBT from the root. This exception is opportunely triggered by a very cheap memory compare instruction. Nevertheless, the whole switching procedure does not require additional locks – i.e. in case of multiple background worker processes, a fine grained latching and status modifications in its meta data are sufficient for synchronization.

4.2.2.4. Defragmentation and Dense-Packing

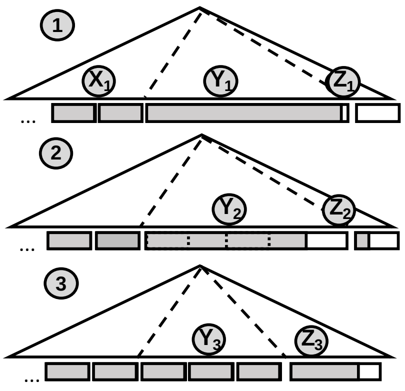
Defragmentation and dense-packing of leaf nodes and immutable upper levels facilitates **(H1)**, **(H2)** and **(H3)** and is a performance critical operation in the regular B$^+$-Tree structure of a PBT. Since a partition match up a sub-tree of a PBT, modifications in the most recent partition typically yield a fill factor of 50% to 80% (as depicted in Figure 4.11a (C_1)) due to random split operations [Gra11]. Objective of defragmentation in PBT is to bring the randomly created leaf nodes in a serializable sequence of optimal read and write I/O aligned dense-packed nodes as necessary – i.e. fill factor of nodes is optimized by blank space minimization, removal of obsolete records and compression techniques; whereas SA is minimized.

Therefore, a naive approach could be to remove obsolete records (Garbage Collection – as outlined in Section 4.4.2), move others across nodes as well as moving nodes across buffer frames, whereas multiple latches are required and node references are updated [Gra11]. Total costs of this procedure depend on the underlying B$^+$-Tree characteristics on structure modifications. Since victim partitions are frequently queried by concurrent transactions, this approach could downgrade overall performance in very traditional lock-assuming B$^+$-Tree implementations.

Assuming a very traditional B$^+$-Tree structure in a PBT as depicted in Figure 4.11a ①. (A_1) depicts immutable persistent partition(s), (B_1) a concealed partition number in the meta data, (C_1) a victim partition with random workload arranged nodes and (D_1) (as well as (D_2) and (D_3)) a modification-bearing most recent partition, which was created by a lock-free partition switch. A very straightforward approach is re-bulk loading required records of the victim partition (C_1), though into another dense partition (B_1), which



(a) Bulk re-insertion approach in a free PBT partition B_1.



(b) Reconciliation of flexibly sized main-memory nodes. [RP22]

Figure 4.11.: Defragmentation and dense-packing approaches in (a) lock-assuming and (b) memory-optimized B$^+$-Tree structures.

is transparent for concurrent transactions. Due to the guaranteed alpha-numeric sort order in the victim partition, the returned records of a partition scan can be sequentially *bulk inserted* into the dense partition by simply overwriting the partition number (depicted in Figure 4.11a ②B_2). Bulk insertions are very beneficial in PBT [Gra03], as records can be successively inserted into unspent leaf nodes of a sub-tree. Since an insert key is known to alpha-numerically succeeding the previous one, a natural append-based behavior is achieved. Consequently, blank space is neither required in nodes nor created as result of random split operations, and records can be densely arranged with an optimal fill factor. B$^+$-Trees typically achieve this behavior on bulk insertions by moving the split point to the end of a full node on insertion. Major benefit of this approach is, that concurrent reading transactions can proceed without interference, since the victim partition C_2 remains unaffected in main-memory until the procedure succeeds, but gets finally erased by a beneficial bulk deletion [Gra11], as depicted in Figure 4.11a ③, when the dense partition B_3 gets visible to concurrent transactions. Since every partition related node is likely to be kept in main-memory and it is an one-time occurring operation, the effort is manageable, however, temporary a considerable memory overhead occurs.

An interesting approach is to leverage the native behavior of main-memory optimized B$^+$-Tree structures. For instance, a lock-free B$^+$-Tree can apply large flexible node sizes (as outlined in Section 3.1.5.2 and depicted in Figure 4.11b). Since binary search costs depend on the number of records (Equation 3.2) and nodes are optimized and considered for main-memory, maintenance of a fixed block size for secondary storage devices is not required. Once a node gets evicted, split operations are performed in order to restore a regular on-disk layout with fixed node sizes in a reconciliation process [Mon21]. In Figure 4.11b, a victim partition $\textcircled{Y_1}$ in a PBT leverages this behavior by allowing a node size as large as the partition itself. At a partition switch, the main-memory mapped flexibly sized node gets reconciled (as outlined in Section 3.1.5.2 and depicted in Figure 4.11b ② $\textcircled{Y_2}$) with the result of dense-packed sequentially arranged leaf nodes $\textcircled{Y_3}$. Whenever the victim partition becomes persistent and evicted, memory mapping of referencing inner nodes is repealed as pointers refer to persistent storage locations, and can be also sequentially written to secondary storage devices. Generally, this approach is very beneficial – especially in case of storage management – since sequentially replicating valid records in a dense partition could cause overheads.

PBT is able to achieve a defragmented dense-packed layout within the structure of any B$^+$-Tree implementation – with negligible effects on payload. The costs of this operation depend on the appropriated features and structure characteristics.

4.2.2.5. Generating Auxiliary Structures

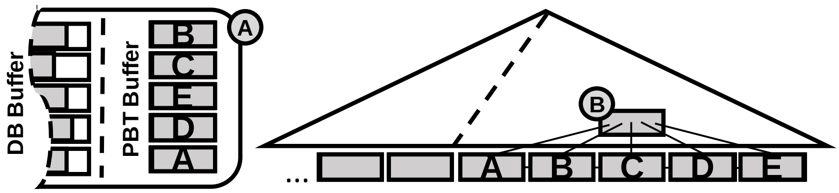
Auxiliary structures, e.g. auxiliary filter structures for data skipping like bloom filter [Blo70], are generated as a natural by-product of the defragmentation process. Since every leaf node of a partition is accessed in the defragmentation process, data-dependent auxiliary structures are incidentally created with marginal costs. Once auxiliary filter structures are created, they can be utilized for data skipping methods (as outlined in Section 4.4.3) – i.e. unnecessary traversal operations are effectively avoided on data, which is located in partitions on secondary storage devices. *These auxiliary structures are very cheap to generate and typically small enough to be located in main-memory.*

4.2.2.6. Beneficial Sequential Write Pattern

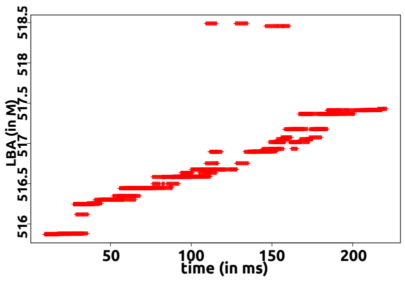
PBT achieves a beneficial sequential write pattern with low WA and SA to secondary storage devices by collecting modifications in a main-memory mapped set of leaves in the PBT-Buffer. A naive strategy to subsequently write the leaves after a partition switch is to simply iterate every buffer frame in the PBT-Buffer and subsequently write the parent inner nodes from common buffer frames (depicted in Figure 4.12a Ⓐ). However, buffer frames are commonly shared for all PBTs in a DBMS and are randomly assigned as a result of the structural modifications within the B^+-Tree structure. Hence, a semi-sequential write pattern with weakly key sorted leaves occur, even if defragmentation is applied (as depicted in Figure 4.12b with a sequence of file extends).

Since leaves are known to be a sequence of key-sorted children of a parent inner node and siblings are well-known in a B^+-Tree, a tree walk guided approach is considerable (as depicted in Figure 4.12a Ⓑ). The PBT is traversed from its root to the leftmost leaf node (**A**) of the dense-packed victim partition. Flushing leaves by principally following the sibling pointers (from **A** to **E** and so on) enables a perfect sequential write pattern (Figure 4.12c from time 0 to 80ms), while the leaves belong to the victim partition. Siblings of leaves are also known in the parent inner node by child reference pointers, whereas this operation is very cache efficient in practice. Subsequently, the parent inner nodes are sequentially written at time 85 to 90ms followed by their parents at 90 to 92ms, respectively.

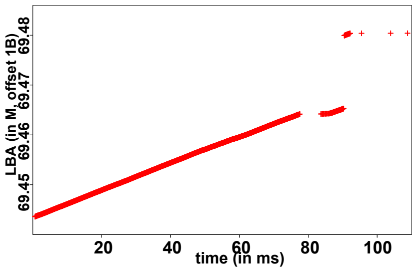
Comparing Figures 4.12b and 4.12c, the tree walk procedure finishes faster, due to the perfect sequential write pattern. Furthermore, with strict correlation in logical block addresses (LBA), everything feasible from DBMS perspective is



(a) PBT flush approaches are based on Ⓐ unsorted PBT-Buffer Frames or Ⓑ the sorted B$^+$-Tree structure.



(b) PBT random buffer frame flush



(c) PBT tree walk flush

Figure 4.12.: Sequential Write of Partition Leaves and Inner Nodes in PBT. Logical Block Address (LBA) offsets and time (in ms) are measured with `blktrace` on a *Samsung 850 Pro* consumer SSD (10MB partition size).

done to leverage Flash secondary storage characteristics – likewise for reading and deleting downstream operations. Hypothesis ***(H2)*** *is confirmed.*

4.2.2.7. Lazy Eviction and flexible assignment of PBT-Buffer Frames

Regular replacement policies in DBMS clean pages by a flush in an eviction operation and downstream replacement with an appropriate page. In case of PBT-Buffer frames, an eviction on flush is utterly inadequate as the effective DBMS buffer cache size would shrink by a significant degree. Moreover, the evicted pages are leaf nodes of the most recent partition, which are frequently queried, e.g. by parallel executing search operations like auxiliary structure creation or time saving index scans.

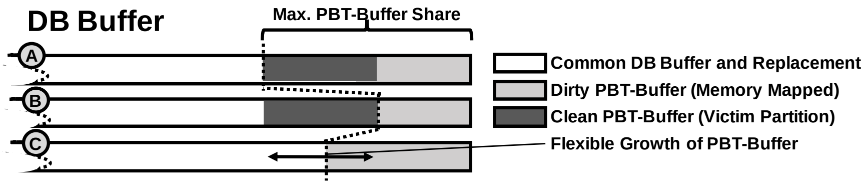

DB Buffer
Max. PBT-Buffer Share
A
B
C
Common DB Buffer and Replacement
Dirty PBT-Buffer (Memory Mapped)
Clean PBT-Buffer (Victim Partition)
Flexible Growth of PBT-Buffer

Figure 4.13.: Flexible PBT-Buffer Share allows cache preserving handover of a clean Victim Partition from Ⓐ the PBT-Buffer to Ⓑ a common database buffer replacement policy and Ⓒ flexible growth up to a maximum PBT-Buffer Share. [RP22]

Flexible PBT-Buffer share preserves the effective buffer cache size and hot data of the victim partition (depicted in Figure 4.13). Whenever a dense-packed victim partition becomes persistent by a sequential write of the PBT-Buffer replacement policy Ⓐ, its clean buffers are passed to the regular replacement policy of the database buffer Ⓑ. In order to sustain a regular replacement policy, statistics of passed buffer frames are adjusted – e.g. in case of most recently used replacement, the usage counter is set to a low average level to avoid immediate eviction. Since the PBT-Buffer share is lower than the allowed maximum threshold Ⓒ, allocations of new leaf nodes are provided by buffer frames of the regular replacement policy. Therefore, the contained and ideally clean page is replaced by a new page and passed under control of the PBT-Buffer, whereby early eviction of modifiable leaves is avoided.

Contrary to other append-based structures, like LSM-Trees, PBT is able to provide a stable main-memory usage with low administrative effort. PBT-Buffer replacement policy with flexible share allows the desired sequential write pattern of immutable partitions without omission of buffer cache efficiency.

4.2.3. Basic Operations

PBT allows functionalities and utilization of interfaces equal to any basic B$^+$-Tree, whereby an integration in many B$^+$-Tree based approaches is feasible. Horizontal partitioning of randomly inserted data enables an append-based

```
typedef struct pbt_cursor {
  void*        search_conditions;
  void*        update_value;
  uint16_t     position;
  ref_t*       node;
  void*        record;
  Partition*   partitions;
  void*        anti_matter_map;
} pbt_cursor_t;
```
Listing 4.2: Sample PBT Cursor Type for Search Operations and Modifications

sequential write pattern, but has also effects on basic operations. In the following, the basic operations are outlined with reference to Listing 4.2 `pbt_cursor_t* cur`.

4.2.3.1. Setting Search Key

Search Keys are crucial for searching a tree structure. B$^+$-Tree structures allow equality as well as range searches. Therefore, search key attribute values are applied with search operators (`cur->seach_conditions`, `lower`, `lower_equal`, `equal`, `greater_equal`, `greater`) in order to define the search key or range. Its first key attribute value is transformed to a partitioned key as outlined in Section 4.2.1.2. Partition numbers are well cached in the *Cached Meta Data* (Section 4.2.1.1) and referenced in `cur->partitions`, whereby the partitioned key is efficiently set to the most recent partition number (*max_pnr*) or any possible valid partition number.

4.2.3.2. Getting Record Key

Index records are stored in alpha-numeric sort order in the tree structure. Regular B$^+$-Trees are able to simply return the uncompressed index record key (`cur->record.key`) on current position as a result of a search operation. PBT relies on partitioned keys (Section 4.2.1.2) for the first key attribute values, hence the partition number is part of the internal key. However, the partition number requires to be transparent to further DBMS modules. Since the regular key is entirely contained in the partitioned key, its attribute value is cheaply returned as an offset without additional processing costs.

4.2.3.3. Getting Record Value

Record values are of various data types in storage and index management – i.e. any value can be stored whereby PBT addresses versatile fields of application. In case of PBT as index management structure, record values are typically short logical or physical references to data tuples in a base table. Since a search operation position the cursor only on '*matter*', i.e. records of type *regular record* or *replacement record*, there is no difference in getting values compared to regular B^{+}-Trees (`cur->record.value`).

4.2.3.4. Equality Search

Equality search is a widely used operation in storage and index management structures for many purposes – e.g. receiving values, checking primary key violation or uniqueness constraints. Tree-based structures search a record by a traversal operation from a root node to a leaf – leading to a logarithmic complexity.

Algorithm 4.1 Interface – Equality Search

```
 1: function EQUALITY_SEARCH($conditions\{attr, operator\}$)
 2: Output: $record\_value$
 3:     Let $cur \leftarrow$ INIT(...)                        ▷ initialize cursor
 4:     SETKEY($cur, conditions$)        ▷ allocate internal partitioned key
 5:     if SEARCH($cur$) then                          ▷ search in the PBT
 6:         $assert$(GETKEY($cur$) $\in conditions$)        ▷ transparent pnr
 7:         return GETVALUE($cur$)                       ▷ return the value
 8:     end if
 9:     return $\emptyset$                          ▷ no record with equal key
10: end function
```

Partitioning in PBT is transparent to further DBMS modules, hence an exemplary *equality search* interface (Algorithm 4.1) is not affected. Internally, PBT search operations pursue an advanced strategy. Partitions are successively searched from the most recent one towards the eldest and lowest numbered for several reasons. First, modifications are appended in the main-memory partition by special record types, which invalidate their predecessors. In order to identify the currently valid record, the invalidation order scheme must be respected. Second, recent partitions are well-cached, hence I/O costs are negligible. Third, equality searches require no special sort order, whereas probably unnecessary pressure of several parallel traversal operations is avoided. Last, additional functionalities, like primary key or uniqueness constraints, imply the existence of only one valid equal key record, whereas unnecessary search effort is saved.

Algorithm 4.2 PBT – Search

```
 1: function SEARCH($PBT\_Cursor\ cur, ...$)
 2: Output: $valid\_record$
 3:     Let $skeys_{part} \leftarrow cur_{\rightarrow}search\_conditions$
 4:     Let $part \leftarrow cur_{\rightarrow}partitions$
 5:     do                          ▷ loop partitions from most recent to eldest one
 6:         SETPARTITION($skeys_{part}, part.pnr$)          ▷ memset pnr in key
 7:         if PARTITION_OFFSET($skeys_{part}$) $\in part.filter$ then
 8:             TRAVERSE($skeys_{part}$)                    ▷ set cursor position
 9:             if $cur_{\rightarrow}record_{\rightarrow}type$ not $\equiv$ Regular Record then
10:                 PUT($cur_{\rightarrow}anti\_matter\_map, cur_{\rightarrow}record$)  ▷ invalidation
11:             end if
12:             if $cur_{\rightarrow}record \in cur_{\rightarrow}anti\_matter\_map$ then ▷ invalidated?
13:                 return NEXT($cur$)                   ▷ move to a valid record
14:             else
15:                 return $TRUE$
16:             end if
17:         end if
18:         ...                         ▷ probably special internal treatments
19:     while $part \leftarrow$ PRECEDINGPARTITION($part$)
20:     return $FALSE$
21: end function
```

In doing so, effort of equality search operations increase by the number of partitions, because in theory every partition has to be traversed – at least until an equal key record was found and no further key is expected. Additional read I/O is leveraged by Flash based secondary storage devices, due to high parallelism and asymmetric read performance. Auxiliary filter structures enable data skipping (as outlined in Section 4.4.3), therefore only a set of necessary partitions is processed and comparably less cheap read I/O is saved (Algorithm 4.2).

Searching and iterating data in partitions involves several records of different types, which indicate an append-based possibility for out-of-place modifications (as outlined in Section 4.2.1.3). Therefore, several records potentially exist in different partitions for one logical data tuple, which form a singly linked list of circumstances. Since partitions are successively processed from the most recent one to the eldest, circumstances are processed in an equal order. However, only one record contains a valid value for a logical tuple – i.e. the most recent (committed) record of a logical tuple is valid[1]. Assuming key uniqueness, an equality search operation can break on the first matching record. Depending on the record type, this has different effects. Record types with '*matter*' included (Regular and Replacement Record) define the current value of a logical tuple. Pure '*anti matter*' (Tombstone Record) defines an invalidation of all logical tuples with an equal key, whereby no matching value is returned.

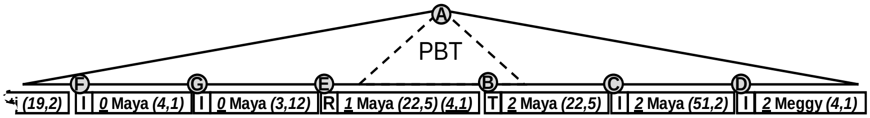


Figure 4.14.: Example Search and Next iteration in PBT. Notation: Regular Records (I), Replacement Records (R), Tombstone Records (T). An Equality Search for 'Maya' result in reference pointer values Ⓒ *(51,2)* and Ⓖ *(3,12)*.

[1]Race conditions require to be treated by the DBMS, e.g. SI in MVCC as outlined in Section 4.2.5.

PBT is designed as a full featuring data and index management structure. Therefore, non-uniqueness require to be covered by the search algorithm and cursor iterator. Since record types identify a circumstance of a logical tuple, invalidation by visited Replacement, Anti or Tombstone Records are collected in `cur->anti_matter_map`. Distinction of respective logical tuples is application dependent, as outlined in Section 4.2.1.3 and 4.2.5.

Records are visited by `search` traversal operations and `next` calls on cursor iterator as depicted in Figure 4.14. The search operation traverses the tree in the most recent partition (2) for a search key '*Maya*' from root Ⓐ to leaf and positions the cursor on the Tombstone Record Ⓑ. Its pure '*anti matter*' 'Maya *(22,5)*' is remembered for later occurrence of '*matter*' and the cursor is moved to the next position Ⓒ. As a result, the cursor is positioned and the value (reference pointer) *(51,2)* can be returned. The `next` operation positions the cursor on the next matching record. Iterating the records in PBT by Algorithm 4.3 moves the position to record Ⓓ, which is not in the scan condition. The partition number is decremented to (1) and the partition gets traversed by the `search` algorithm from Ⓐ and positioned at record Ⓔ. Replacement Records contain '*matter*' (Maya *(22,5)*) as well as '*anti matter*' (Maya *(4,1)*). Therefore, the latter is remembered for later occurrence. However, the former is already invalidated by the Tombstone Record Ⓑ and gets skipped. Finally, partition (0) is traversed from root Ⓐ to leaf and the cursor is positioned at record Ⓕ. Once again, the index record was already invalidated by Ⓔ and the cursor is moved to Ⓖ, which can be returned. *As demonstrated, PBT features full storage and index management functionalities for equality search.*

4.2.3.5. Range Search

Range search is a very efficient operation in trees unlike to other structures. With a marginal logarithmic overhead of a traversal operation, any required range span of search keys is obtained by simply iterating records, since they are already arranged in key sorted order. Therefore, inclusive or exclusive range spans are defined by a left and a right search key as well as associated

Algorithm 4.3 PBT – Next

```
 1: function NEXT($PBT\_Cursor\ cur, \ldots$)
 2: Output: $valid\_record$
 3:     Let $skeys_{part} \leftarrow cur_{\rightarrow}search\_conditions$
 4:     loop
 5:         ITERATE($cur$)                                   ▷ move to next position
 6:         if not GETKEY($cur$) $\in skeys_{part}$ then
 7:             break
 8:         else if $cur_{\rightarrow}record_{\rightarrow}type$ not $\equiv$ Regular Record then
 9:             PUT($cur_{\rightarrow}anti\_matter\_map, cur_{\rightarrow}record$)        ▷ invalidation
10:         end if
11:         if not $cur_{\rightarrow}record \in cur_{\rightarrow}anti\_matter\_map$ then ▷ invalidated?
12:             return $TRUE$
13:         end if
14:     end loop
15:     $cur_{\rightarrow}partitions \leftarrow$ PRECEDINGPARTITION($part$)
16:     return SEARCH($cur$)
17: end function
```

operators.

PBT introduces an advanced level arrangement of search keys, due to its partitioned keys. Partitioning is transparent to further DBMS modules, however, it might affect the arrangement of a result set. In keeping with the example in Figure 4.14, lets take again '*Maya*' as left key and broaden the query for a right key 'Meggy' in an inclusive range. In performing similar to the equality search, the range search first returns Ⓒ 'Maya *(51,2)*', continuing with Ⓓ 'Meggy *(4,1)*' – since it is equal to the right key – and finally return Ⓖ 'Maya *(3,12)*'. Since 'Meggy' lexicographically follows 'Maya', the arrangement in the result set is affected. There are three possibilities to deal with it. First, the query does not require any sorting criteria at all. Indexes are applied and utilized for many purposes, e.g. if joining data of different tables by foreign keys. Assuming an equal development in location of data over time (e.g. in append-based DBMS), a PBT index returns very fast a subset of co-located record reference pointers of data tuples in base tables. These are commonly batch-wise fetched from base table pages and added to the result set, while PBT traverses the preceding partition. A well parallelizable query processing with batch-wise semi-sorted tuples is enabled. Second, if an alpha-numerically sorted result set is required in the query predicates, a downstream merge sort is applied. It mostly might be an appropriate method in DBMS indexing, since indexes are version-oblivious and therefore sorting is not guaranteed. Furthermore, the whole result set is required to enable a downstream merge sort, wherefore it is rather a possibility for internal operations. Last, for the presented cursor notation, an online merge sort is applied. Auxiliary filter structures are tested in order to narrow the probable number of partitions. For each remaining partition, a separate internally managed cursor is enabled, and the partitions get traversed in parallel. High parallelism in Flash based secondary storage devices is leveraged and also appropriate, since definitively every remaining partition must be checked. Record keys get compared from the most recent partition to the eldest. If a record gets returned or a record is pure '*anti matter*', the internally managed cursor is simply iterated to the next position. Thereby, '*anti matter*' in a more recent partition causes a preceding partition cursor to iterate, too. In keeping with the example in Figure 4.14, three internally managed cursors are applied. The Tombstone Record Ⓑ causes the cursor of Replacement Record Ⓔ also to increment. However, Ⓔ also contains '*anti matter*' and therefore the last internally managed cursor at Regular Record Ⓕ is also incremented. Within one comparison, the cursors are moved from (Ⓑ,Ⓔ,Ⓕ) to (Ⓒ,$\emptyset$,Ⓖ) and Ⓒ gets returned. As a result, the internally managed cursor is moved from Ⓒ to Ⓓ. Since the record key of Ⓖ is lexicographically preceding Ⓓ, it gets returned next and its internally managed cursor is set to $\emptyset$. Finally, Ⓓ is returned and the internally managed cursor is set to $\emptyset$. The query is answered in the correct alpha-numeric sort order (Ⓒ Maya *(51,2)*, Ⓖ Maya *(3,12)* and Ⓓ Meggy *(4,1)*). *PBT introduces a location and time preserving partitioning pattern in append-based DBMS with (semi) sorted record keys. Furthermore, an online merge sort enables strict alpha-numeric sorted iterable result sets.*

4.2.3.6. Sorted Insert

Datasets are frequently maintained. Alpha-numerically key-sorted structures, like B$^+$-Trees, maintain a strict arrangement of records and therefore they require to move records in nodes, split over nodes and massively write already persisted data to secondary storage – which adversely result in random write patterns and massive WA.

PBT focus insertions in a main-memory mapped fraction of the tree structure – the most recent partition in the PBT-Buffer – in order to tackle these issues. Since this area acts like a main-memory B$^+$-Tree, many lock free maintenance optimizations become highly valuable, e.g. as outlined in Section 3.1.5.2.

Extension of a dataset is enabled by the insertion of *Regular Records* in the most recent partition. Therefore, the partitioned key (first attribute value) of the insert key is extended by the current most recent partition number (`max_pnr` in PBTMeta Data) within the `setKey` function. A regular B$^+$-Tree insert operation with a root to leaf traversal is performed in order to find

Algorithm 4.4 PBT – Insert

```
 1: function INSERT($PBT\_Cursor\ cur, ...$)
 2: Output: $success$
 3:     Let $skeys_{part} \leftarrow cur_{\rightarrow}search\_conditions$
 4:     Let $success \leftarrow FALSE$
 5:     do
 6:         SETPARTITION($skeys_{part}, max\_pnr$)          ▷ memset pnr in key
 7:         TRAVERSE($skeys_{part}$)                        ▷ set cursor position
 8:         ACQUIREWRITEACCESS($cur$)
 9:         if $max\_pnr$ = GETPARTITION($skeys_{part}$) then
10:             $cur_{\rightarrow}record \leftarrow$ FORMREGULARRECORD($cur$)
11:             $success \leftarrow$ INSERTINLEAF($cur_{\rightarrow}record, cur_{\rightarrow}position$)
12:         end if
13:         RELEASEWRITEACCESS($cur$)
14:     while not $success$
15:     return $success$
16: end function
```

the insert location and probably acquires write access. Before the record gets inserted in a leaf node, the partition number in the partitioned key is compared with the `max_pnr` in PBT Meta Data. If both are still equal, the record is regularly inserted in the leaf node. Though, `max_pnr` might gets incremented by a concurrent partition switch operation. In this case the partitioned key of the insert record is modified to match the newly created most recent partition number and the traversal operation is repeated, since the formerly set partition became immutable. In principle, just an unchecked 'blind insertion' is outlined, i.e. a record is sufficiently inserted without any constraints. However, additional constraints are guaranteed by performing regular preceding search operations as needed. *Partitioning in PBT enables highly concurrent insertions in main-memory and therefore it guarantees an append-based sequential write pattern with low WA.*

4.2.3.7. Value Update

Data tuples evolve over life time. B^+-Trees modify records in place. If the record search key is not affected by a modification, the value is simply changed. However, if the record search key is modified, the current record becomes invalidated (ghost record) and the new record is inserted at its designated location within the tree structure. Both cases result in modifications allover the tree structure, yielding a random write pattern and high WA to secondary storage devices.

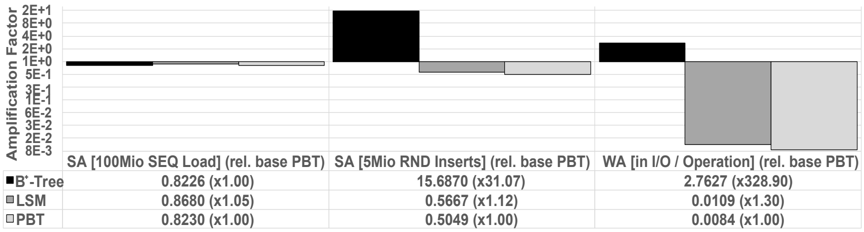
PBT handles modifications to a logical tuple by insertions of different record types – describing circumstances over life time. Insertions of Anti and Replacement Records underlie an equal constraint to insertions of Regular Records, i.e. they are directed to the most recent partition. Equal to regular B^+-Trees, modifications affect the value and / or record search keys. In the former case, a Replacement Record with distinguishing values is sufficient. Since the record search key is equal to its preceding circumstance (except the partition number), a downstream search operation is able to assign its '*anti matter*' to the invalidated circumstance record, as already known from the equality search example in Figure 4.14. If applicable, still modifiable records in the most recent partition are updated in place, equal to regular B$^+$-Trees, however the invalidation scheme must be respected. In the latter case, a combination of Anti Record and Replacement Record is applied. Generally, Anti Records contain the record search key (with most recent partition number) and value of their preceding circumstance record. Replacement Records contain the new record search key and value. *Therefore, PBT is able to modify logical tuples over life time without violating the beneficial sequential write pattern to secondary storage devices.*

4.2.3.8. Record Delete

Deletions describe the end of tuple life cycles. Storage and index management structures adopt different approaches. Records of deleted tuples get immediately removed or invalidated for later maintenance. PBT apply Tombstone Records for out-of-place invalidation. Tombstones are pure '*anti matter*' and contain the record search key of the latest circumstance record with '*matter*' (except the partition number) as well as its record value. Tombstone Records are inserted in the most recent partition in order to meet the append-based nature of PBT. If applicable, still modifiable records in the most recent partition are probably removed (Regular Records) or transformed to Tombstones (Replacement Records), since preceding circumstances require to remain invalidated. In keeping with the example in Figure 4.14, the search algorithm is able to recognize an invalidation of Replacement Record Ⓔ by the Tombstone Ⓑ. *Since Ⓔ still invalidates Regular Record Ⓕ, all maintained circumstances of the logical tuple remain invalidated without affecting further logical tuples like Ⓖ in indexing non-unique datasets.*

4.2.4. Cost Model

PBTs are based on the structure of ubiquitous B$^+$-Trees. Hence, operations in PBT are subjecting a logarithmic complexity, too. Decisive factor is the height h of a tree as defined in Equation 3.1. Familiar variables are the average fan-out F of inner nodes, fill factors of inner f_i and leaf nodes f_l and the required number of leaves L_N to store N records. However, there are major influencing factors. First, size of record search keys in PBT is marginally increased and therefore L_N is adversely affected. If prefix truncation is enabled (as outlined in Section 4.2.1.2), this factor is decoupled from the number of records a leaf node potentially contains. As a result, slightly increased L_N is usually negligible (compare SA (100Mio SEQ Load) in Figure 4.15). Second, the number of records N is increased by the number of modifications (replacements R) to logical tuples – at least temporary as an intermediate state. Compression of large values might counteract, however it is a native behavior of out-of-place updating structures and necessary to achieve a beneficial sequential write pattern. Since it primarily affects L_{N+R}, which are probably rather located on cheap secondary storage devices, there are practically no effects on the cache efficiency. Furthermore, the height h is practically never increased, due to its logarithmic complexity. Last, average fill factor of immutable inner nodes f_i and leaves f_l are guaranteed to be optimal, due to defragmentation on partition switch (outlined in Section 4.2.2). Moreover, this is also valid for mutable leaves in the valuable PBT-Buffer and referring inner nodes, if flexible page sizes are applied (as outlined in Section 3.1.5.2). In both cases, f_i and f_l are approximately equal to



	SA [100Mio SEQ Load] (rel. base PBT)	SA [5Mio RND Inserts] (rel. base PBT)	WA [in I/O / Operation] (rel. base PBT)
■B⁺-Tree	0.8226 (x1.00)	15.6870 (x31.07)	2.7627 (x328.90)
▨LSM	0.8680 (x1.05)	0.5667 (x1.12)	0.0109 (x1.30)
□PBT	0.8230 (x1.00)	0.5049 (x1.00)	0.0084 (x1.00)

Figure 4.15.: Comparison of B$^+$-Tree, LSM-Tree and PBT in SA after bulk load, SA after random insertion and resulting WA.[1] [RP22]

[1]Depicted SA factors are based on the actual dataset / update-set size, WA quantifies the write I/O per operation. Amplification factors in brackets indicate PBT as baseline. All structures enable prefix truncation and snappy compression. Merges are disabled for comparability.

1, due to the low fraction of mutable nodes. In a regular B$^+$-Tree, the fill factors are among 0.5 and 1, since insertions cause node splits – e.g. in the experiment in Figure 4.15, five million random insertions in a defragmented and read optimized B$^+$-Tree cause the structure to excessively increase in size by a factor of 15.7 to the actually inserted data, since nodes get split and the average fill factor shrinks. Generally speaking, the height of a PBT h_{PBT} is under no circumstances larger than the height of a B$^+$-Tree h for *massive* amounts of data. However, Equation 3.1 is adapted to PBT by consideration of modifying replacements (R) in the number of required leaf nodes ($L_N \rightarrow L_{N+R}$) and optimal fill factors of nodes ($f_i, f_l \in [0.5; 1] \rightarrow f_i, f_l \simeq 1$):

$$h_{PBT} = \lceil \log_{F \times f_i} \frac{L_{N+R}}{f_l} \rceil \; + \; 1 \; , \quad F \neq 1 \; , \quad L_{N+R} > 1 \; , \; f_i, f_l \simeq 1 \tag{4.1}$$

Furthermore, search effort in PBT depends on the number of partitions P – i.e. the number of relevant partitions for the search operation $P_s \subseteq P$. Relevant partitions stem from many influencing factors:

- *Uniqueness constraints in equality search operation* enable PBT to break partition processing on the first matching record.
- *High selectivity of search predicates and datasets in partitions* allow the utilization of auxiliary filter structures.
- *Accuracy of auxiliary filter structures* (compare Section 4.4.3, Chapter 5).
- The *applied search algorithm*. Whereas equality search operations with high selectivity rely on sequential processing and read I/O prevention, low selectivity queries leverage high parallelism in Flash secondary storage devices in multiple parallelized traversal operations.

Consequently, parallelizable processing effort is a function of the absolute number of partitions P and the resulting relevant partitions P_s, denoted as $f_{(P)}$. Moreover, auxiliary probe costs on up to P filter structures C_{filter_probe} are considered in P_f:

$$s_{PBT} \approx P_f \times C_{filter\_probe} + f_{(P)} \times \log_2(N+R) \ \ , f_{(P)} \in [1;P_s], P_f \in [0;P-1] \quad (4.2)$$

Overall processing costs increase as a result of partition maintenance. As outlined in Section 2.1.1, highly parallelized and decentralized processing units are expected. Since traversal operations in PBT are parallelizable for each partition, PBT profits from trends in processing units and additional costs are expected to be manageable. With regards to Equation 2.1, read I/O costs must be higher rated. Costs of Equation 3.4 are amplified by the number of partitions P – i.e. $f_{(P)}$ to be precisely – for search operations in PBT. Certainly, cache efficiency of inner nodes p_{ic} is increased in PBT ($p_{ic_{PBT}}$) compared to evolving structure of regular B$^+$-Trees. Comparing fill factor of inner nodes f_i in Equations 3.1 and 4.1, contained data is reduced up to 50% in B$^+$-Tree, whereas more inner nodes are required to cover underlying data. This behavior is experimentally evaluated in Figure 4.16, where a read-only workload is performed upon different number of modifications. Throughput of PBT is higher in this read-only workload, since it is more efficient to search several partitions with optimally filled inner nodes rather than caching partially filled ones. A very similar effect can be observed

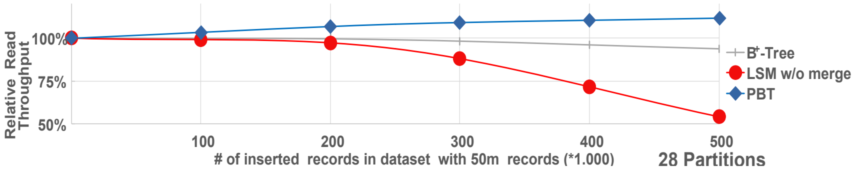


Figure 4.16.: Read-only workloads on tree structures after increasing number of randomly performed modifications.[1]

[1]Relative read-only throughputs are respectively normalized to structures' initial throughput (on Consumer SSD, 3 hours steady state random reads). PBT not only preserves read-only throughput despite from increasing number of filter-protected partitions, but as well improves due to increased $p_{ic_{PBT}}$ in more recent partitions and break conditions in the equality search algorithm (compare Section 4.2.3.4). B$^+$-Trees suffer from partially filled nodes as result of random insertions and LSM-Trees suffer from fragmentation in separate components.

for LSM-Trees (for comparability merge operations are disabled), whereas an equal fragmentation exist as in PBT, however, root nodes are neither commonly cached nor its logarithmic capacity is leveraged – i.e. several root nodes exist with a very low fill factor. Generally, the increased cache probability of inner nodes ($p_{ic_{PBT}} > p_{ic}$) by commonly cached separator keys allows at least comparable read throughput:

$$s_{I/O_{PBT}} \approx f_{(P)} \times R_{I/O} \times (\lceil \log_{F \times f_i} \frac{L_{N+R}}{f_l} \rceil \times (1 - p_{ic_{PBT}}) + (1 - p_{lc}))\,,\; p_{ic_{PBT}} > p_{ic} \gg p_{lc} \tag{4.3}$$

Since modern Flash based secondary storage devices provide high parallelism for sequential and random read I/O (as outlined in Section 2.1.2) and traversal operations are known to be parallelizable for each partition, PBT leverages characteristics of Flash in the search algorithm. Resulting read I/O could be increased, but well captured by characteristics of secondary storage devices.

Write I/O is carrying much more weight (approx. $R_{I/O} \times 10$ and probably resulting erases $R_{I/O} \times 100$), as outlined in Section 2.1.2. PBT treats modifications to the dataset as insertions of different record types in the most recent partition. Equal to regular B$^+$-Trees, the insert location is initially determined by a search operation, however, only the most recent partition is traversed ($f_{(P)} = P_s = 1$). Since leaves are memory mapped anyways as well as inner nodes in this area have a very high cache probability (up to 100% due to techniques introduced in Section 4.2.2), following is valid for this operation: $p_{ic_{PBT}}, p_{lc} = 1$. Hence, no read I/O costs to secondary storage devices occur for the insert operation, but related operations like uniqueness constraint checks or evaluation of preceding circumstances can cause $s_{I/O_{PBT}}$ as previously described. Write I/O in PBT is caused on sequential write step of a partition switch operation. Partitions P contain regular records of newly inserted data (N^*) and replacements to already existing records (R) in several leaf nodes $L_{(N^*+R)_P}$. Furthermore, a number of covering inner nodes are processed, which are a sum function of the leaf nodes $g_{(L_{(N^*+R)_P})}$ with an optimal fill factor f_i and fan out F. Therefore, the issued background $W_{I/O}$ per modifying operation $i_{I/O_{PBT}}$ is a fraction of the affected nodes of a partition (as depicted in Figure 4.15 WA (in I/O per Operation)):

$$i_{I/O_{PBT}} \approx W_{I/O} \times \frac{L_{(N^*+R)_P} + g_{(L_{(N^*+R)_P})}}{(N^* + R)_P} \tag{4.4}$$

Worst case modification costs in PBT are updates to the value of a key attribute. In this case, a search operation with the read costs of $s_{I/O}$ is required as well as the insertion costs of two records for invalidation of the old key and validation of the new key. Since PBT leverages $s_{I/O_{PBT}}$ as outlined above (compare Figure 4.16), and the much more expensive $W_{I/O}$ costs are a much less factor in $i_{I/O_{PBT}}$ than in $i_{I/O}$ (regular B$^+$-Tree cause at least one $W_{I/O}$ in Equation 3.5), PBT outperforms its competitors B$^+$-Tree and LSM-Tree.

Even though massive amounts of data and high rate continuous insertion workloads are assumed, each modification is only written once on a sequential persistence operation of all leaf nodes of a partition. Therefore, the WA of every single uncompressed node is equal to 1. This circumstance is valid whilst SA of invalidated records is manageable on cheap secondary storage devices and $s_{I/O_{PBT}}$ is reasonably consistent:

$$WA_{PBT} = 1 \tag{4.5}$$

Compression techniques, like prefix truncation or snappy (as applied in Figure 4.15), enable PBT to shrink WA and SA even further, so that factors of less than 1 are achieved. B$^+$-Trees exhibit expensive structural modifications in order to maintain the alpha-numerical sort-order, yielding massively higher SA and expensive WA (compare Figure 4.15). Contrary, PBT applies horizontal partitioning with few SA and WA and benefit from structural properties and modern hardware characteristics on search operations. Thereby

PBT outperforms LSM-Trees, due to its caching behavior on traversal operations and slightly improved SA and WA. PBT exhibits beneficial structural properties compared to its competitors.

4.2.5. Summary

A simple and broadly applicable horizontally partitioned write-optimized storage and index management structure has been designed in PBT – on base of the structuring pattern introduced by [Gra03]. Horizontal partitioning allows to leverage structural properties and modern techniques in B^+-Trees in a very hot fraction of the tree and ultimately enable a beneficial sequential write pattern with low WA. **(H1)**, **(H2)** and **(H3)** have been theoretically and experimentally evaluated and have shown major benefits in WA, sequential write pattern and SA – compared to its competitors B^+-Tree and LSM-Tree. Performance characteristics of fully integrated PBT implementations are evaluated in benchmark workloads in Chapter 6.

However, PBT is version-oblivious – i.e. visibility check operations rely on separate DBMS modules. Circumstances over tuple life cycle are well represented by record types, however, since PBT is able to return probably visible tuple candidates, *anti matter* is not able to invalidate preceding circumstances for several possible transaction snapshots and additional work is necessary. Therefore, probably memory-exhausting mapping structures or additional reads to secondary storage devices is required, yielding cache inefficiency and principally RA to the base table main data store, even though PBT comes with good base capabilities to cope with amplified pressure of *rolling entry points* (introduced in Section 2.4) in version record maintenance.

4.3. Version-Awareness with Multi-Version Partitioned BTree

Multi-Version Concurrency Control (MVCC) is a popular concurrency control protocol in modern DBMS and K/V-Stores. In combination with *Snapshot Isolation* SI, each transaction is able to operate in its individually calculated snapshot of data. Therefore, several versions of one logical tuple exist – each

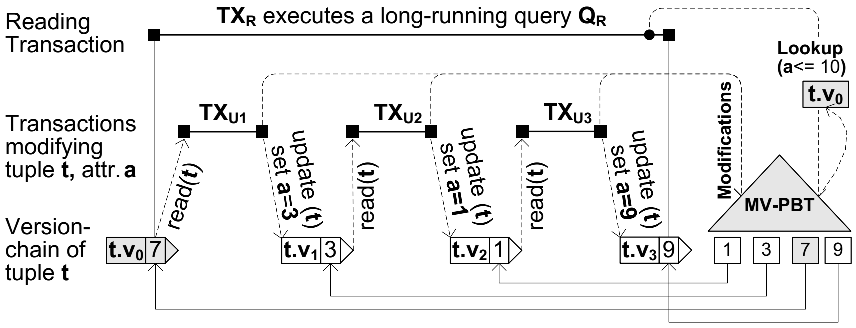


Figure 4.17.: MV-PBT maintains several version records of modifying transactions ($TX_{U\{1,2,3\}}$) and returns only the respectively visible one to each transaction snapshot (TX_R and $TX_{U\{1,2,3\}}$). [RVGP20]

valid for a different period in time. As outlined in Section 2.3 and listed in Table 2.1, storage manager beneficially maintain physical representations of timestamped tuple version records and new-to-old version ordering with one-point invalidation, since this approach enables directly accessible version records and a sequential write pattern.

However, since additional access paths are typically version-oblivious, the version chain in the base table main store requires to be processed for visibility checking anyways – yielding massive RA on base tables. For instance, in Figure 4.17, a long-lasting transaction TX_R performs a query Q_R. In the meantime, tuple t is frequently modified by concurrent transactions (TX_{U1}, TX_{U2}, TX_{U3}) and successor versions $t.v_1$, $t.v_2$ and $t.v_3$ are created upon the base version $t.v_0$, which is related TX_R's transaction snapshot. A version-oblivious index cause 4 random read I/O in base tables in order to identify the related version.

Especially in case of HTAP with long-lasting analytical processing and frequent concurrent updates, such version management schemes yield excessive version chains and massive amounts of transient version records [MBL17] (illustrated in Figure 4.17) – as high as several hundred millions in real systems [LSP+16].

PBT is already able to inexpensively maintain circumstances of logical tuple life cycles by operation-dependent record types. This behavior correlates very well with the maintenance of version records (with well known properties from Table 2.1). Multi-Version Partitioned BTrees (MV-PBT illustrated in Figure 4.17) fully enable multi-version storage and index management on base of PBT within a single structure, by:

- strict out-of-place version record maintenance and arrangement
- annotation of validation and/or invalidation timestamp information
- prevention of massive RA by an Index-Only Visibility Check

The featuring objectives of MV-PBT are formulated in following hypotheses:

Hypothesis 4 (H4)
MV-PBT is able to serve as multi-version store on modern hardware.

Hypothesis 5 (H5)
MV-PBT amplifies selectivity for version records and prevent from massive RA on base tables by an Index-Only Visibility Check.

In doing so, MV-PBT still retain horizontal partitioning and write-optimization characteristics of PBT, defined in **(H1)**, **(H2)** and **(H3)** and features multi-version capabilities in **(H4)** and **(H5)**.

4.3.1. Applying Multi-Version Record Types

MV-PBT likewise features modifications to logical tuples over their life cycles by inserting different index record types introduced in Section 4.2.1.3 in the most recent partition. Thereby it is ensured, that MV-PBT is a fully featuring multi-version storage as well as index management structure with native non-uniqueness and multi-attribute treatment. In order to provide multi-version capabilities, the record types are annotated with the transaction timestamp of the inserting transaction, either for validation of the new version record and/or invalidation of the predecessor version record (depicted in Figure

4.18.Ⓕ). Thereby, an (index-only) visibility check is able to determine the visibility of each version record to a transaction snapshot (described in Section 4.3.6). Multi-version record types are generally declared by *flags*[1] in the record header (Figure 4.18.Ⓐ) and are described as follows:

Regular Records describe the begin of a tuple life cycle, hence there is no predecessor version record. It consists of a Ⓑ *partition number*, one or several Ⓒ *key attribute values*, a Ⓓ *record value* as well as a Ⓕ *transaction timestamp* for validation. The transaction timestamp is obtained by the inserting transaction. Key attribute values and the record value are defined by the schema, application and operation on data. For instance, the physical reference to the version record in base table is stored in the Ⓓ validation record value.

Replacement Records describe the modification of a *value attribute* of a predecessor version record, which is referencing a logical tuple. It consists of a Ⓑ *partition number*, one or several Ⓒ *key attribute values*, one Ⓓ or two Ⓔ *record value(s)* and a Ⓕ *transaction timestamp* for its validation by the modifying transaction as well as logical invalidation of its predecessor version. Likewise, the partition number is the most recent partition number. Since MV-PBT requires to conserve predecessor version for a period of time,

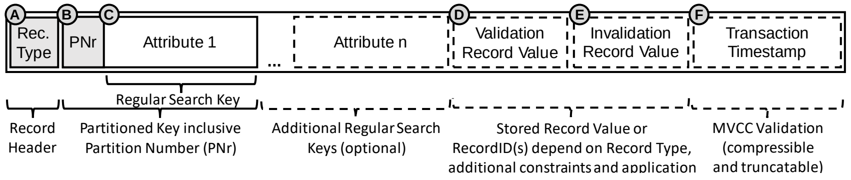

A
B
C
D
E
F
Rec. Type
PNr
Attribute 1
...
Attribute n
Validation Record Value
Invalidation Record Value
Transaction Timestamp
Regular Search Key
Record Header
Partitioned Key inclusive Partition Number (PNr)
Additional Regular Search Keys (optional)
Stored Record Value or RecordID(s) depend on Record Type, additional constraints and application
MVCC Validation (compressible and truncatable)

Figure 4.18.: Basic structure of a Multi-Version Record Type. Dashed elements are optional or truncatable.

[1]According to Section 4.2.1.3, 2 bits in the record header are sufficient, one as '*replacement indicator*' and another as '*pure anti matter*' – i.e. Regular Records '00', Replacement Records '10', Anti Records '11' and Tombstone Records as '01'.

altering a principally modifiable existing predecessor in place is impossible, therefore maintenance of several *Replacement Records* of a logical tuple within a single partition requires special arrangement conventions, as outlined in Section 4.3.5. Unlike to version-oblivious PBT, MV-PBT is able to perform an (index-only) visibility check, hence it must be possible to retrace version chains without indirection layers. A Ⓓ validation record value stores the modified value and the Ⓔ (truncatable) invalidation record value stores the predecessor value – e.g. in case of index management, the physical reference of the new tuple record and the predecessor tuple record in base table are stored.

Anti Records describe the invalidation of their predecessor version on modification of *key attribute values* and are utilized in combination with *Replacement Records*, since different record search key replacements are not sufficient to invalidate predecessor versions. It consists of a Ⓑ *partition number*, Ⓒ one or several *key attribute values*, at most Ⓔ one *record value* and a Ⓕ *transaction timestamp* for invalidation. Key attribute values and (truncatable) invalidation record value are obtained by the invalidated predecessor version.

Tombstone Records describe the end of a tuple life cycle on deletion, hence no successor version is possible. It consists of a Ⓑ *partition number*, Ⓒ one or several *key attribute values*, at most Ⓔ one *record value* and a Ⓕ *transaction timestamp* for invalidation. Equal to Anti Records, key attribute values and (truncatable) invalidation record value are obtained by the invalidated predecessor version.

4.3.2. Version-Aware Record Sizes and Compression Techniques

Functionality in MV-PBT increase by the cost of larger record sizes. Consequently, less records are potentially stored per leaf node. However, additional storage costs are diminished, due to several compression and truncation techniques.

Apply Timestamp Offsets. Transaction timestamps are compressible. Partitions are frequently switched, which is why they are active for a limited number of transactions. Applying a minimum transaction timestamp to each partition in its meta data allows a compressed delta representation in each record – e.g. an 8 byte transaction timestamp can be restored from singly stored meta data and 2 byte delta per record for 2^{16} transactions.

Invalidation Value Truncation. Invalidation record values are truncated, if MV-PBT is applied as storage management structure or a primary key or uniqueness constraint is defined, since the search key value is sufficient to retrace version chains. Auxiliary secondary indexes without uniqueness constraints require invalidation records anyways, however, physical references are relatively small.

B$^+$-Tree Compression Techniques. Dense-packing techniques (outlined in Section 4.2.2) are applied on partition switch – i.e. prefix truncation and optimal fill factor with required records are guaranteed.

Transaction Timestamp Truncation. While dense-packing, transaction timestamps, which are obsolete for visibility checking, i.e. visible to every active transaction, are completely truncated.

As a result, space requirements in MV-PBT only slightly increase compared to PBT, however, one-point invalidation by multi-version record types allow (index-only) visibility checking in version-aware basic operations.

4.3.3. Logical Version Chain in evolving MV-PBT

In B$^+$-Tree structures, records are not fixed to a determined physical reference, since an alpha-numeric sort order is maintained. Hence, in MV-PBT, physical references to a predecessor version record in new-to-old ordered version chains (as outlined in Section 2.3.1.2) are not an option.

MV-PBT introduces a logical new-to-old version chain approach – i.e. version records are rather logically built on query execution as part of the search operation (outlined in Section 4.3.5) than maintaining physical references to predecessor versions.

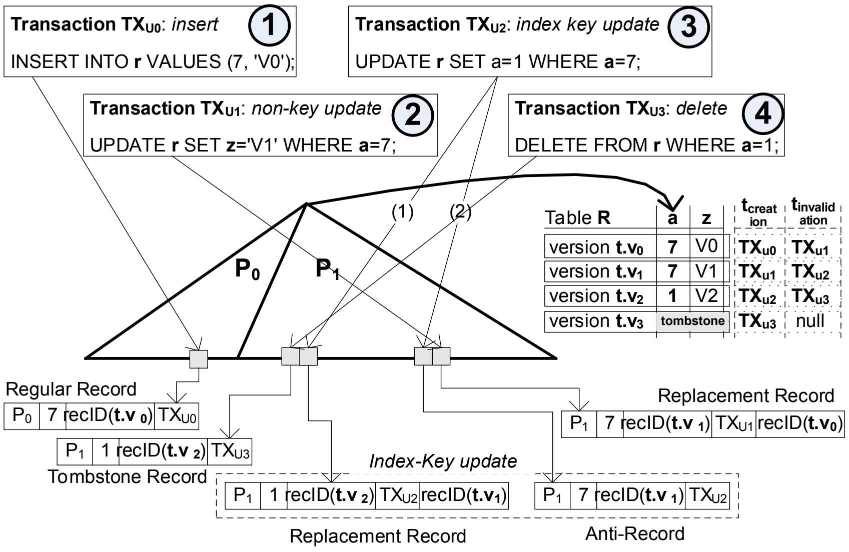
Physical references to predecessor versions are generally maintained by a *record id* consisting of *block* and *slot offsets*, beginning with an *entry point*. However, this presupposes that predecessor versions remain unchanged at a specific location. In a B$^+$-Tree structure, exact block and slot offsets are unlikely to be stable, since they maintain a lexicographically sorted set of records. Actually, it is sufficient to know predecessors record search key to find them by a cheap traversal operation in the tree structure. Assuming the transaction processing purpose of version record maintenance in MVCC with SI applying DBMS, it is rather useful to identify snapshot related version records (*lateral entry*) than identifying whole version chains of one logical tuple, which is more relevant to administrative processes, like garbage collection.

Logical referencing of version chains in MV-PBT offers several advantages. First, even if evicted immutable partitions are persistent, modifications to a logical tuple occur independent from partition switching, and indeed probably more frequent, wherefore predecessor versions are unlikely to be located at their final block and slot offset, since records are sorted by their search key. An indirection layer from logical tuple to volatile version record physical references is neither desirable nor necessary. Second, logical tuples are searched in a tree structure based on their search key attributes. Thereby the relation to a transaction snapshot of a version record is relevant and not its entire history or *entry point* (except for modifications). Version records relation to transaction snapshots is cheaply calculated (outlined in Section 4.3.6) as part of the search operation (outlined in Section 4.3.5) due to the ordering properties of MV-PBT. Last, version chain information is logically comprised and can be restored by scanning partitions – i.e. administrative operations are incidentally able to interlink version chains on partition switching or statistics generation.

Logical new-to-old version chains in MV-PBT are based on several information. *Record types* (i.e. the underlying operation), the *search key attribute values*, *validation* and *invalidation record values* and the *transaction timestamps*. For linkage between version records, their relation to a logical tuple must be identified by unique information, however, a separate virtual identifier (VID [GPHB17]) is not required. Record types, which modify a logical tuple, contain an *invalidation record value*, which is equal to their predecessors *validation record value*. Moreover, either the *search key attribute values* are equal to its predecessor or an *Anti Record* with an equal *invalidation record value* as well as *transaction timestamp* exist. Ordering of committed *transaction timestamps* is known, wherefore the most recent version record of a logical tuple is identified as *entry point* and already accessed predecessors can be appended. Additional *primary key* or *uniqueness constraints* simplify the linkage process, since the *search key attribute values* already imply unique logical tuple assignment and *invalidation record values* are truncated.

Example Logical Version Chain Reconstruction. For instance, in Figure 4.19 several operations on a logical tuple are performed as described in Section 4.3.5. ① Transaction TX_{U0} creates a logical tuple by inserting a *Regular Record* with the search key 7 in MV-PBT as additional access path with references to a base table R. An equal key *Replacement Record* is modifying the tuple by generating a new version 1 in ② transaction TX_{U1}. ③ Transaction TX_{U2} performs a search key update by inserting an *Anti Record* with for invalidation of ② and a *Replacement Record* with the new search key attribute value 1. Finally, the tuple gets deleted by the insertion of a *Tombstone Record* in ④.

In this example, transactions are known to have an ascending order, so for new-to-old ordering, the most recent transaction number TX_{U3} describes the invalidation of version record with the *entry point* to the logical tuple on reference $t.v_2$. It is found first by traversing and iterating from P_1 to P_0. Proceeding the scan lead to its invalidated predecessor version with an equal search key attribute value and validation of $t.v_2$, but also an invalidation of the version record reference $t.v_1$. Since there is no further equal search key



Transaction TXU0: insert
1
INSERT INTO r VALUES (7, 'V0');
Transaction TXU2: index key update
3
UPDATE r SET a=1 WHERE a=7;
Transaction TXU1: non-key update
2
UPDATE r SET z='V1' WHERE a=7;
Transaction TXU3: delete
4
DELETE FROM r WHERE a=1;
(1)
(2)
P0
P1
Table R
a
z
tcreation
tinvalidation
version t.v0 | 7 | V0 | TXu0 | TXu1
version t.v1 | 7 | V1 | TXu1 | TXu2
version t.v2 | 1 | V2 | TXu2 | TXu3
version t.v3 | tombstone | TXu3 | null
Regular Record
P0 | 7 | recID(t.v 0) | TXU0
P1 | 1 | recID(t.v 2) | TXU3
Tombstone Record
Replacement Record
P1 | 7 | recID(t.v 1) | TXU1 | recID(t.v0)
Index-Key update
P1 | 1 | recID(t.v 2) | TXU2 | recID(t.v1)
P1 | 7 | recID(t.v 1) | TXU2
Replacement Record
Anti-Record

Figure 4.19.: Logical Version Chain Maintenance in MV-PBT. [RVGP20]

record with this version record reference value, there must be one *Anti Record* with a different search key. This *Anti Record* is identified by scanning P_1 until the search key attribute value 7. The occurrence of this record implies a modification of the search key attribute value from 7 to 1 between tuple versions $t.v_1$ and $t.v_2$ by transaction TX_{U2}. Continuing the iteration lead to the *Replacement Record* inserted in ② by means of validation of tuple version $t.v_1$ and invalidation of $t.v_0$ in transaction TX_{U1}. Finally, the *Regular Record* is found in P_0, which is describing the oldest version record. *The new-to-old ordered version chain can be reconstructed, based on information within the MV-PBT, with an equal result to the physically maintained version chain in Table R.*[1]

[1]According to Section 2.3 and Table 2.1, the assumed version chain is maintained by physically materialized and new-to-old ordered version records with one-point invalidation model in the base table, however design and implementation details possibly differ but have no impact on the comprised information. Hence, a general base table design is illustrated in Figure 4.19. MV-PBT represents equal information in a logically maintained version chain.

4.3.4. In-Partition Version Record Ordering Convention

Within search operations it is necessary to identify *invalidation* of a version records before their *validation*, therefore MV-PBT applies version record ordering conventions.

New-to-old version record ordering must be guaranteed by version-aware basic operations. This behavior can be achieved in different ways (depicted in Figure 4.20).

Leveraging Reversed Succession of Partitions. A naive approach could be to simply continue the partitioning scheme by the maintenance of several main-memory partitions – i.e. a separate partition for each new successor version record of a logical tuple (depicted in Figure 4.20.①, partition numbers 1, 2 and 3). While searching all partitions from the most recent one to the lowest numbered, the version records are processed in the desired new-to-old ordering scheme and the version chain is natively reconstructible (Ⓑ,

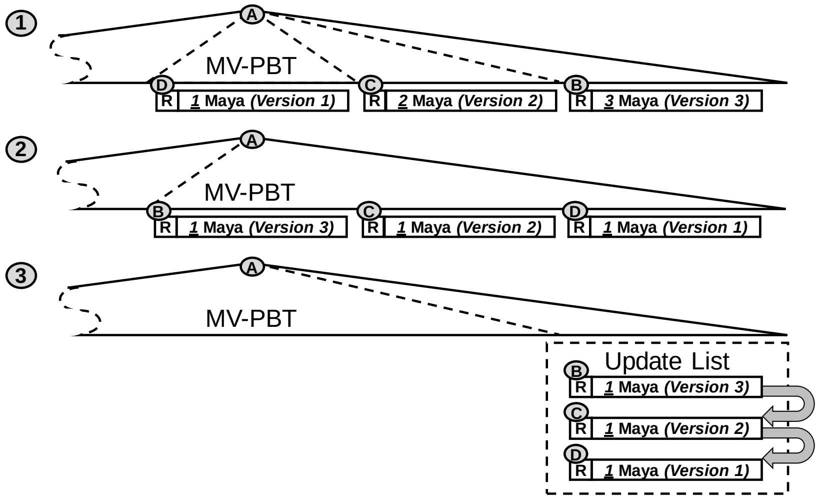


Figure 4.20.: Version Record Ordering features correct representation of Version Chain.

Ⓒ, Ⓓ). However, load imbalances yield massive occurrence of additional sparsely filled main-memory partitions. Since main-memory partitions are only protected by cheap partition fence keys (min/max key in each partition, outlined in Section 4.4.3) as well as minimum transaction timestamps of comprised version records and no further more accurate auxiliary filter structures, resulting fragmentation is fully sweeping on operation costs, due to multiple traversal operations (respectively from Ⓐ to Ⓑ, Ⓒ and Ⓓ). As a worst case scenario, for instance, a tuple with a lower and an other with a higher sorted key are modified X times whilst other tuples remain constant yield up to $X + 1$ main-memory partition traversal operations for any operation on other tuples, since cheap partition fence keys become ineffective – even if every version record is located on a single leaf node. In order to avoid multiple traversal operations, excessive garbage collection of transient version records and comprised reorganization would be necessary.

Alpha-Numeric In-Partition Ordering. A more intelligent way to arrange successor version records in MV-PBT is an in-partition ordering convention of searching and inserting operations (depicted in Figure 4.20.②). Thereby, equal search key successor records are simply placed to be found first on search operations. For instance, a MV-PBT inserts new records leftmost to other equal search key records on modification, whereby an equal ordered search criteria result in first accessing most recent version records of any type (only one traversal from Ⓐ to Ⓑ and iterating to Ⓒ and Ⓓ) – contrary ordered search criteria need special treatment, whereby a reverse iterator requires to access the leftmost equal search key record first. Even if this approach is applicable in most B^+-Tree variants without precautions, it is not without disadvantages. Range search operations need to process several version records which are not visible to its transaction snapshot, since temporal evolution is intermingled with actual data. For instance, if Ⓑ *Version 3* in Figure 4.20.② is visible to a range search transaction snapshot (e.g. `TX`$_{V3}$`: SELECT (*) FROM 'table' WHERE 'attribute' BETWEEN 'Maya' and 'Meggy';`), Ⓒ *Version 2* and Ⓓ *Ver-*

sion 1 are processed (but not returned) as part of the `next` operation (compare Algorithm 4.3 and in addition Section 4.3.5), even though they are already invalidated by Ⓑ *Version 3*.

Leveraging In-Memory Update Lists. Separation of temporal evolutionary version records and actual data can be achieved by leveraging modern B^+-Tree techniques [Gra11; Mon21]. Equal key version records of one logical tuple build a update list, which is referenced in a main-memory leaf node of the most recent partition (depicted in Figure 4.20.③). These update lists are purged as part of the reconciliation process in the partition switch dense-packing phase (Section 4.2.2), whereas the regular disk layout of still required version records is restored. For immutable persistent partitions the version ordering conventions ① and ② are applied.

However, this approach brings major benefits. First, binary search costs on traversal of the most recent partition are reduced to a function of the actual dataset, since comparisons are reduced to one per logical tuple in this partition. Obviously, modifications to a key attribute of a logical tuple require separate update lists. Nevertheless, the number of comparisons in a binary search are principally independent from the number of version records. Second, iterators in range searches are able to skip new-to-old arranged predecessor version records of a logical tuple in the most recent partition as if the visibility check identified the visible version record to the transaction snapshot. For instance, if Ⓑ *Version 3* (in Figure 4.20.③) is visible to a transaction snapshot, neither Ⓒ *Version 2* nor Ⓓ *Version 1* are processed by the iterator. Therefore, processing effort of range search operations is independent from the number of predecessor version records. Obviously, successor version records require to be processed anyways in a new-to-old version ordering as part of the visibility check. Third, in combination with other modern B^+-Tree techniques, like flexible page sizes (outlined in Section 3.1.5.2), this approach massively reduces maintenance operational effort, e.g. record movement, since it is delayed to a single reconciliation process on partition switch in MV-PBT. Last, version records are co-located in update lists, whereby visibility checking is simplified and accelerated. Other approaches might intermingle equal key records of different logical tuples. Challenges in in-partition version ordering conventions are well covered by modern B$^+$-Tree techniques, which are leveraged by MV-PBT.

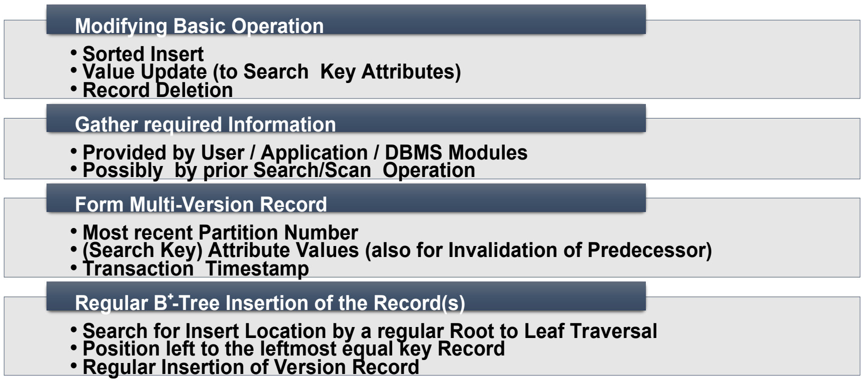
4.3.5. Version-Aware Basic Operations

MV-PBT combines basic PBT operations and desirable storage management properties from Table 2.1 in Section 2.3 for storage as well as index management. This means for the former that MV-PBT adopts a new-to-old ordering approach (Section 2.3.1.2) of physically materialized version records (Section 2.3.1.1) with logical out-of-place update scheme and one-point invalidation model (Section 2.3.1.3). *Similarly, MV-PBT as index management structure applies physical references to version records in an equal arrangement for additional access paths.*

Subsequently, basic operation extensions are outlined on base of standard PBT. Therefore, the cursor based notation introduced in Section 4.2.3 can be applied without cautions. However, a `pbt_cursor` is extended by `snapshot_information` of the DBMS for native visibility checking (outlined in Section 4.3.6) by equality and range search operations. The required transaction timestamp information is provided by the utilized record types (outlined in Section 4.3.1) by modifying operations.

4.3.5.1. Modifying Operations

In MV-PBT, every modifying basic operation is performed by an out-of-place insertion of a new version record in the most recent partition with in-partition ordering convention (outlined in Section 4.3.4), since preceding version records are probably still required by concurrent transactions. Version record types are formed by the *basic operation*, its *transaction timestamp* and user or application defined *search key* and *value attributes*. Thereby it is necessary, that logical version chains (Section 4.3.3) are unambiguously interpretable by the search algorithm. Based on the equal preconditions of



Modifying Basic Operation

- Sorted Insert
- Value Update (to Search Key Attributes)
- Record Deletion

Gather required Information

- Provided by User / Application / DBMS Modules
- Possibly by prior Search/Scan Operation

Form Multi-Version Record

- Most recent Partition Number
- (Search Key) Attribute Values (also for Invalidation of Predecessor)
- Transaction Timestamp

Regular B$^+$-Tree Insertion of the Record(s)

- Search for Insert Location by a regular Root to Leaf Traversal
- Position left to the leftmost equal key Record
- Regular Insertion of Version Record

Table 4.1.: Modifying Operations likewise perform Insertions of different timestamped Record Types and build a logical Version Chain.

modifying operations (sorted insert, value update, value update to search key attributes and record deletion), their similarities, i.e. *regular insertions* of individually generated records in the most recent partition, are concisely listed in Table 4.1 and their particular characteristics are outlined in the following.

Sorted Insert. This operation is performed on the beginning of a logical tuple life cycle, wherefore a *Regular Record* type is applied. User or application defined predicates are passed in the appropriate search key and value fields. The most recent partition number is utilized in the partitioned key and the current transaction timestamp is added to the version record. A root to leaf traversal is performed and positioned left to the leftmost equal key record in the most recent partition and the *Regular Record* is inserted by a regular B$^+$-Tree insert operation. For instance, a *Regular Record* is inserted into partition P_0 with the search key attribute value 7 and physical reference to version record $t.v_0$ as value by transaction TX_{U0} (compare Figure 4.19 ①).

Value Update. Updates define modifications to attribute values of logical tuples. *Replacement Records* are possibly inserted without searching for predecessor versions (blind insertion) or are generated on base of a result set of a prior search operation. Both approaches are applied in modern K/V-Stores. For instance, it could be designed for fast storage of modifications to the dataset and allows to logically overwrite records by simply inserting a new equal key record out-of-place (blind insertion), however, it also supports to update a set of qualified logical tuples, whose information is gathered by a prior scan of auxiliary access paths and complex predicates. Also, the most recent partition number is utilized in the partitioned key, the appropriate search key and value fields are filled by the gathered information and the current transaction timestamp is added to the version record. Based on the application, its *Invalidation Record Value* is filled or gets truncated. Finally, the *Replacement Record* is inserted at its designated position (compare Figure 4.19 ②).

Value Update to Search Key Attribute Values. In case of modifications to the search key attribute values of a logical tuple, the predecessor version additionally must be invalidated by an equal search key *Anti Record* in the most recent partition. *Validation Record Values* are not applied in this record type, since it is pure *anti matter*, however, the *Invalidation Record Value* is filled with its predecessor record value or gets truncated. The current transaction timestamp is added to the version record, which is inserted at its designated position (compare Figure 4.19 ③).

Record Deletion. Deletions describe the end of a logical tuple life cycle, which is performed by the insertion of a *Tombstone Record*. It is formed by the most recent partition number, its predecessor key attribute values, a probably truncatable *Invalidation Record Value* and the current transaction timestamp. Finally, it is inserted at its designated position (compare Figure 4.19 ④).

4.3.5.2. Query Operations

MV-PBT search operations are principally equal to its base PBT variant, but with the addition of an included visibility check. Therefore, the result set of MV-PBT equality and range search operations only include visible version records of logical tuples instead of record candidates.

Partition Selection and Cursor Positioning. MV-PBT performs root to leaf traversals on a (sub-)set of partitions by manipulating the partition number in the partitioned key. The (sub-)set of partitions gets selected by probing auxiliary filter structures (outlined in Section 4.4.3). Partitions are logically processed from the most recent to the lowest numbered one, however, based on the selectivity of query predicates, sequential or merge sort approaches are performed (as outlined in Section 4.2.3).

Generating Snapshot Result Sets. Version ordering is crucial for multi-version search operations in one-point invalidating new-to-old ordering approaches. Accuracy is guaranteed by invalidation via *Anti Records* on modifications to the search key attributes (Section 4.2.1.2), in-partition ordering conventions (Section 4.3.4) and the logical sequence of searched partitions from most recent to the lowest numbered one (Section 4.2.3). Search operations iterate the version records of traversed (sub-)sets of partitions and operate based on the visibility to its transaction snapshot. '*Invisible*' version records are simply skipped, '*visible*' invalidation is added to the `anti_matter_map` introduced in Section 4.2.3 and '*visible*' and not invalidated version records are added to the result set.

For instance, an *Anti Record* is visible to a transaction snapshot in an equality or range search, even if its predecessor's search key attributes have been modified and predecessor versions can be excluded on base of the `anti_matter_map`. However, its successor version is not directly known by the *Anti Record* (Section 4.3.3). Indeed, by altering the search predicates it is still possible to find and return its successor version to the result set, if its timestamp is within the transaction snapshot – independent from its predecessor's invalidation by the *Anti Record*. On the other hand, an *Anti Record* is not visible to a transaction snapshot of a prior started range search and becomes simply skipped without adding its information to the `anti_matter_map`. *This concept of invalidation is valid within one as well as across several partitions.*

HTAP Query Processing Characteristics. HTAP workloads comprise of short-running OLTP as well as long-lasting OLAP queries. In this context, the presented search algorithm behaves different for both workload categories.

Version records near the version chain's *entry point* are likely to be related to OLTP transaction snapshots in a new-to-old version ordering. In MV-PBT, partitions that comprise these version records are likely to be accessed first, since partitions are processed from most recent to the lowest numbered one.

However, the outlined behavior would not be beneficial in case of concurrently executed long-lasting OLAP queries. Since OLTP transactions create several successor versions and partitions as well, downstream commands in OLAP could require to process several partitions without related version records. Minimum transaction timestamp of comprised version records are a lightweight auxiliary filter structure (compare Section 4.4.3), which is able to skip partitions that are created after a transaction started – i.e. partitions with higher minimal transaction timestamp than the executing transaction are entirely skipped. A *lateral entry* to the related version record is achieved with minimized search and traversal costs, *hence MV-PBT optimally supports both characteristics of HTAP queries.*

In order to identify excluded and included version records to a transaction snapshot, a visibility check is required. MV-PBT is able to natively perform visibility checks as part of its search operation while iterating and probing version records within the search criteria.

4.3.6. Index-Only Visibility Check

MV-PBT facilitates amplified load of timestamped version records treatment for storage as well as index management with the goal of native multi-version visibility check support – i.e. the result set of equality and range search operations in MV-PBT is natively reduced to the version records, which are visible to a transaction snapshot.

4.3.6.1. Incidental Multi-Version Visibility Check in MV-PBT

Modifications in MV-PBT to a logical tuple are treated as insertion of a version record of different types (outlined in Sections 4.3.1, 4.3.5) in the most recent partition. Thereby it is ensured that version records of one logical tuple are processed in a consistent succession, since a new-to-old version ordering (Section 2.3.1.2) with one-point invalidation (Section 2.3.1.3) is applied. However, within logically maintained (Section 4.3.3) and processed version chains (Section 4.3.5) of logical tuples, search operations need to determine the visibility of each version record to its transaction snapshot.

Visibility Checks fit in with MV-PBT Search Operations. MV-PBT natively supports usual visibility checks within search operation iteration processes. Similar to invalidated records by *anti matter*, transaction snapshot's unrelated version records are skipped on iteration, wherefore the visibility check operation (Algorithm 4.5) fit in with this process in `search` (Algorithm 4.2 lines 9 to 12) and `next` (Algorithm 4.3 lines 8 to 11) operations. In order to identify relevant validating or invalidating version records, visible transaction timestamps to a snapshot of transaction $TX_{CurrentTxId}$ require to be known by the DBMS. Generally, in MVCC with SI, already committed transactions before transaction $TX_{CurrentTxId}$ has started as well as $TX_{CurrentTxId}$'s inherent modifications belong to $TX_{CurrentTxId}$'s transaction snapshot – i.e. version records of concurrently executed or aborted[1] transactions are unrelated

[1]Transactions are possibly aborted and rolled back by user request or the concurrency control protocol, however, inserted version records remain existent until they are garbage collected.

Algorithm 4.5 MV-PBT Visibility Check

```
function VisibilityCheck( $Cursor\ cur$)
Input: $Cursor$ position information
Output: $visibility$ to transaction snapshot
    ▷ linkage information of version record (Section 4.3.3)
    Let $recID \leftarrow$ getVChainFields($cur_{\rightarrow}record$)
    if queuedForGC($cur_{\rightarrow}record$) then                ▷ (Section 4.4.2)
        return $INVISIBLE$
    end if
    if not precedes($cur_{\rightarrow}record_{\rightarrow}ts, CurrentTxId$) or isConcurrent($cur_{\rightarrow}record_{\rightarrow}ts, CurrentTxId$) then
        return $INVISIBLE$
    end if
    if $ts_{anti} \leftarrow$ get($cur_{\rightarrow}anti\_matter\_map, recID$) then
        $assert$(precedes($cur_{\rightarrow}record_{\rightarrow}ts, ts_{anti}$))   ▷ (Section 4.3.4)
        checkForGC($cur_{\rightarrow}record$)                     ▷ (Section 4.4.2)
        return $INVISIBLE$
    end if
    if containsAntiMatter($cur_{\rightarrow}record$) then     ▷ (Sec. 4.2.3, 4.3.5)
        put($cur_{\rightarrow}anti\_matter\_map, recID, cur_{\rightarrow}record_{\rightarrow}ts$)
    end if
    if containsMatter($cur_{\rightarrow}record$) then
        return $VISIBLE$
    end if
    return $INVISIBLE$
end function
```

$TX_{CurrentTxId}$'s transaction snapshot.

Background: MVCC Transaction Snapshot. In order to calculate the snapshot state of transaction $TX_{CurrentTxId}$, its inherent timestamp (*CurrentTxId* in Algorithm 4.5) and *transaction numbers of concurrently executed transactions* are captured at the beginning of a transaction. Moreover, the DBMS keeps track of *aborted transactions*. Possibly additional information is considered for performance reasons, like the lowest numbered uncompleted concurrent transaction or conclusive record flags for invisibility[1] (Algorithm 4.5 line 6). *By this means, a consistent logical succession of transaction numbers is provided and potentially related and visible version records are identified by their creation timestamp.*

Determine Version Record's Visibility. Solely by the *record information* (`Cursor cur→record`) and existent *snapshot state information*, an usual visibility check is incidentally performed (compare Algorithm 4.5 line 9). Logical succession of committed transactions, *CurrentTxId* as well as *transaction numbers of concurrently executed transactions* are known by the snapshot state. Version records with unrelated transaction timestamps are skipped by a negative visibility check (`INVISIBLE`), i.e. a record is unrelated to $TX_{CurrentTxId}$'s transaction snapshot, if `cur→record→ts` not precedes `CurrentTxId` or `cur→record→ts`'s transaction concurrently committed after $TX_{CurrentTxId}$ started.

Nevertheless, *one-point invalidation* cause record invalidations to be performed out-of-place, which are potentially related to a transaction's snapshot. MV-PBT's *version record ordering* (Section 4.3.4) in the *logically* maintained *version chain* (Section 4.3.3) facilitate a consistent collection of related invalidations (*anti_matter_map*), whenever a record is accessed. Potentially related '*anti matter*' is first iterated by the *search algorithm* (Section 4.3.5) and added to *anti_matter_map* in Algorithm 4.5 lines 17 to 18 – i.e. its unique *record id* is `put` as key and its *transaction timestamp* as value to the *anti_matter_map*. Subsequently accessed '*matter*' is probed for invalidation in the *anti_matter_map* in Algorithm 4.5 line 12. Unrelated '*anti matter*' to a transaction snapshot is already skipped by the incidentally performed visibility check in Algorithm 4.5 line 9. *By this means, only* `VISIBLE` *records with 'matter' are added to a transaction snapshot's result set.*

MV-PBT is natively able to incidentally perform visibility checks as part of

[1]Records can be queued for garbage collection, whenever an obsolete record (invisible to any active transaction snapshot) is checked for visibility (compare Algorithm 4.5 line 14 and Section 4.4.2)

its search algorithm. Necessary information is entirely contained on access to version records in order to compare their visibility to a transaction snapshot. MV-PBT enables preferable append-based storage characteristics listed in Table 2.1 and Flash leveraging search operations, which yield an alpha-numerically sorted result set of version records, hence **(H4)** *is entirely confirmed.*

4.3.6.2. Amplified Selectivity of Indexes in DBMS applying MVCC

Most popular traditional index structure in DBMS is the version-oblivious ubiquitous B^+-Tree of which records reference the *entry point* of a version chain in base tables. It is only capable to return a set of tuple version candidates, wherefore the integrity of its result set is guaranteed, however, visibility checks and potential sorting requirements require additional information spread along several base table blocks. As a result, its native capabilities are limited and maintenance costs are amplified, due to multi-version requirements.

MV-PBT is able to return an alpha-numerically sorted result set of qualified version record attribute values and physical references to base table records while leveraging characteristics of modern hardware technologies. These capabilities yield optimal selectivity of queried result sets and minimized RA on base tables, wherefore MV-PBT is highly qualified as indexing structure in DBMS applying MVCC and SI. Benefits of version-aware indexing with MV-PBT as additional access path to data in base tables are as follows.

Cache Efficiency in Index-Only Visibility Checks. Index records are generally much smaller than version records in base tables, since only indexed attribute values are stored. Furthermore, records that correlate in search key attribute values and creation time are co-located and beneficially processed for both types of HTAP workload characteristics (compare Section 4.3.5). Hence, probing in a small set of searchable sorted index records for visibility is more cache efficient than probing randomly spread large sized tuple version records.

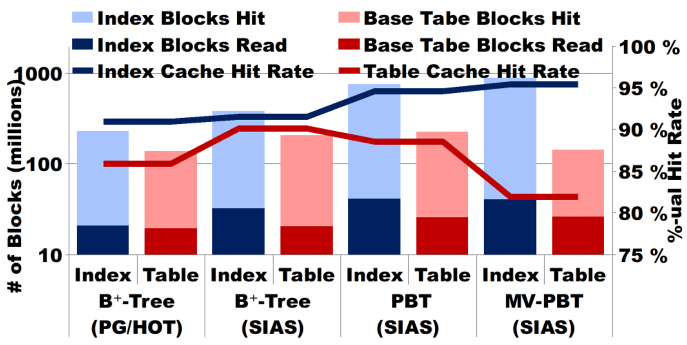
Index-Only Scans. Version-aware indexes allow direct return of queried attribute values without checking their visibility in base tables. Result sets are created upon indexed attributes, whereby RA on base table can be completely avoided.

Minimizing Successive Read I/O. Once the set of partitions is traversed, relevant tuple version records are directly accessible without the need of an indirection layer or processing randomly spread tuple version chains. The number of successively read base table blocks is optimized.

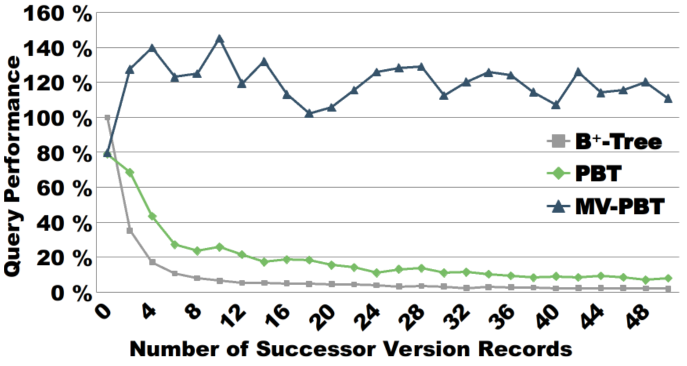
Direct Access to Relevant Version Record. Update-intensive datasets are saturated by tuple versions that are not related to current transaction snapshot. Performance robustness for long version chains of MV-PBT is a basic prerequisite for analytical processing in HTAP.

Experimental Evaluation. In order to proof outlined characteristics of MV-PBT as additional access path, an experimental evaluation in an OLTP scenario (depicted in Figure 4.21a) as well as in an HTAP-like micro scenario (depicted in Figure 4.21b) is given.

In the first scenario, *PostgreSQL 9.0.4* (baseline with an old-to-new ordering scheme and co-located version records in base table (`HOT`)) and *PostgreSQL with SIAS* (new-to-old version ordering and one-point invalidation model) and respectively applied *B^+-Trees*, *PBT* or *MV-PBT* are configured with a fair buffer size of 600MB (since cache hit rates are higher than 80%) for a TPC-C-like benchmark loaded with a dataset of 1500 warehouses. Read I/O and cache hits are measured for an equal number of transactions for base table and index nodes. First observation is made by comparing the baseline with SIAS (Figure 4.21a). Indexing effort in SIAS is amplified by the new-to-old ordering scheme with a *rolling entry point* and yield a larger set of index records, hence the number of accessed index nodes increase compared to the baseline. Furthermore, the visibility check in the append-based base



(a) Cache Hit and Block Read Statistics in an OLTP Workload.



(b) MV-PBT facilitates robust Search Performance Characteristics.

Figure 4.21.: Potentials in Version-Aware Indexing.

table storage require successive block accesses to solve the version chain contrary to HOT, where version records are co-located by the cost of random write I/O. Combined effects increase the block accesses on index nodes by 40% and on table nodes by 60%, thus additional accesses are well cached, due to TPC-C's skewed workload. *However, potentials of SIAS are restricted by index maintenance and successive reads of visibility checks.*

PBT as version-oblivious write-optimized index yield doubled node accesses, due to unqualified partitions for data skipping. However, increased index usage is intended and accesses are well cached by the PBT-Buffer, wherefore additional read I/O indicates a slightly diminishing cache probability in immutable preceding partitions and base tables, due to lessened common database buffer share. Reads on index nodes in version-aware MV-PBT behave similar to PBT with slightly increased number of accesses, due to larger uncompressed version record sizes. Likewise, block read I/O on base tables is increased, due to the MV-PBT-Buffer, however, absolute block accesses decrease to an equal amount as in the co-located HOT version management scheme, which indicates direct access to the related tuple version record and optimized selectivity of the result set. *Even in case of short version chain lengths in an OLTP workload, MV-PBT is able to massively reduce accesses to base table, due to optimized selectivity. Accesses could be further reduced by building projections based on indexed attributes what is not depicted, due to limitations of the query processing engine.*

Query execution characteristics of long-lasting analytical query processing in HTAP are considered in Figure 4.21b. For simplicity, a YCSB workload is configured to perform updates and scans on a user table to generate a background load. Qualified measurements are frequently performed every 30 seconds by an equality search. The version chain length is concurrently increased by creation of successor versions belonging to the designated tuple.

By means of new-to-old version ordering, the long-lasting equality search needs to process the version chain from its entry point to the designated version record for the introduced indexing structures. Measurements are proportionally depicted to B^+-Trees initial query performance. B^+-Trees initially perform well, however decrease rapidly as the version chain grows. Successively accessing randomly spread base table pages lead to a massive performance drop, even in case of short version chain lengths. *PBT generally exhibit a similar trend, however, query performance benefit from reduced occupancy of the secondary storage device, due to beneficial write patterns.*

MV-PBT exhibit robust performance characteristics[1] – independent from the version chain length, since visibility checks on small sized, temporally

[1]Exceptional performance degeneration in case of 0 successor versions (Figure 4.21b) indicate initial lower cache probability of inner nodes, due to 20% smaller common database buffer / inclusive MV-PBT-Buffer.

co-located, laterally entered and searched index version records avoid successive access to randomly spread tuple version information in append-based base table storage by cheaply amplified selectivity in MV-PBT.

MV-PBT natively optimize selectivity of result sets in additional access paths by cost-efficient index-only visibility checks. This is accomplished in a structure, which leverages characteristics of modern hardware technologies, wherefore MV-PBT is highly qualified as additional access path in DBMS applying MVCC with SI. **(H5)** *is entirely confirmed.*

4.3.7. Summary

MV-PBT natively leverages query features of PBT in order to perform (index-only) visibility checks based on horizontally partitioned timestamped record types. Append-based storage management coincide with multi-version record maintenance, whereby MV-PBT features beneficial capabilities to serve as multi-version store on modern hardware technologies and **(H4)** is confirmed. Moreover, **(H5)** is confirmed, since native visibility checks enable an optimized selectivity of accurately queried result sets, wherefore MV-PBT is an appropriate indexing structure in DBMS applying MVCC and SI.

4.4. Workload Adaptiveness and Optimizations

PBT and MV-PBT are temporally evolving storage and index management structures with horizontal partitioning and out-of-place modifications aiming for a beneficial sequential write pattern as well as optimized WA. Even though this is a very desirable feature for load consumption of modifying operations, queries suffer from immoderate fragmentation and SA. *However, evolved fragmentation is not necessarily the final horizontal partitioning.*

Physical in-place modifications, movements or deletions of version records are practically prohibited for basic operations, due to the concepts and algorithms of PBT and MV-PBT – especially as if the comprising partition becomes immutable. *Admittedly, insertions of Replacement Records are feasible for reorganization. Hence, following Hypotheses are formed:*

Hypothesis 6 (H6)
Reorganizations allow reduced search costs without violating basic concepts and algorithms of (MV-)PBT.

Hypothesis 7 (H7)
Entirely replaced partitions by sparsely created and defragmented successor partitions become obsolete and are opportunely truncatable.

Hypothesis 8 (H8)
Auxiliary filter structures cheaply enable data skipping methods and are applicable in (MV-)PBT.

Subsequently, potentials of reorganizations are outlined in the contexts of query optimization by adaptive data reorganization and indexing **(H6)** and space reclamation **(H7)**. Moreover, a detailed cost-benefit analysis of data skipping methods **(H8)** is given.

4.4.1. Adaptive Data Reorganization and Indexing

Horizontal partitioning is a result of the workload dependent growing structure of a MV-PBT. Querying fragmented structures might suffer from increased costs, especially in case of range searches. Moderate fragmentation is well covered by auxiliary filter structures, cached upper levels and the characteristics of secondary storage devices, e.g. asymmetric fast and parallelizable random reads. However, excessive fragmentation adversely influence the performance of query operations whenever the limits of auxiliary filter structure probes, caching or read I/O performance is exceeded.

State-of-the-Art. LSM-Trees overcome this issue by frequently performed merge operations in order to reduce fragmentation. Unfortunately, this approach massively increase WA, since already persisted and still valid data records are written multiple times for frequent reallocation – different merge approaches might vary in their characteristics (outlined in Section 3.1.3). *Contrary, MV-PBT aims for minimized WA.*

Applying Learned Approaches. A further approach is to reinsert necessary version records as by-product of frequently executed queries – e.g. by inserting a Replacement Record in a dedicated memory mapped caching partition, which is (almost) never persisted but frequently evaluated and reorganized for practical benefits like increased cache hit ratio. It can be improved, since frequently accessed records are probably intermingled on nodes with rarely accessed records. Even if this is a very workload adaptive approach, several precautions have to be considered. First, version record ordering must be ensured for still accessible version records by in-partition ordering conventions and logical partition succession, even if successor versions are not visible to a maintaining transaction snapshot. Second, necessary version records must be predictable with different forecasting techniques, e.g. based on heuristics for a skewed workloads. Malfunctioning prediction yield inefficient maintenance costs, cache inefficiencies and additional search costs. Third, even if break conditions in equality search algorithms are met, result sets of range search operations require to be entirely contained in order to save additional traversals. Last, maintenance of memory mapped caching partitions does rarely result in sustainable reorganization. Although this technique requires additional research, it is an interesting approach for adaptive reorganization, especially in case of larger main-memory volumes and hot/cold separation of data. *However, required learned and forecasting techniques are not in scope of this thesis.*

Applying Cached Partitions (CP). Based on the concept of caching frequently accessed version records by the insertion of Replacement Records, *Cached Partitions (CP)* are introduced. CP are a cyclically created persistent comprehensive key-sorted view on the most recent version records of a subset of immutable preceding partitions. CP are probably indicated by '*type*' in PBT Cached Meta Data (Section 4.2.1.1) and comprise only records of type

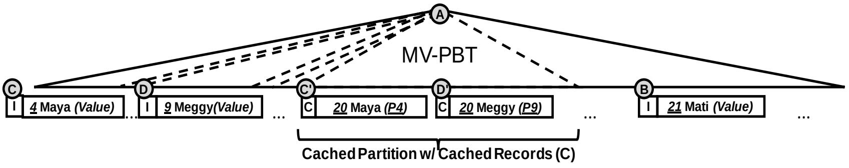


Figure 4.22.: Cached Partitions are space-efficient and sorted Internal Partial Indexes to a Subset of preceding Partitions.

Cached Record. CP fit in the partition ordering scheme and can completely organize the access to contained subset of partitions, due to the consistent set of related version records, hence it is an appropriate approach for data skipping mechanics in horizontally partitioned datasets.

Internal Partial Indexing with CP. Cached Partitions (CP) cover a subset of randomly grown preceding immutable partitions, which are still relevant and accessible for querying. In Figure 4.22, for instance, 20 partitions $(0-19)$ are covered by a CP with the number 20. Comprised alpha-numerically sorted records are related to most recent preceding version record of each logical tuple within the covered subset of partitions. Referenced most recent version records could be of any introduced record type. Records in CP enable searchable and space-efficient referencing of one relevant version record – i.e. CP feature internal partial indexing in (MV-)PBT.

Special Role: Cached Record Type. In CP, records are of type *Cached Record* (denoted as type 'C' in Figure 4.22). They are not intended to replace predecessor version records, however, they underlie similar properties like *Replacement Records* as outlined in Section 4.3.1. Nevertheless, introduced '*Validation Record Values*' feature the reference to the related record version and *Invalidation Record Values* only indicate logical linkage information. If possible, *Invalidation Record Values* are truncated, *Transaction Timestamps* are truncated anyways, since this information is maintained at the referenced version record. Cached Records in CPs constitute a space-efficient key-sorted view on existent and accessible version records.

Record Value Trade-Offs. Cached Records are not intended to replace predecessor version records, however, they feature fast access to stored information in horizontally partitioned datasets. This could be achieved by simply storing the original record value in a Cached Record. CPs would completely replace covered preceding partitions and control fragmentation, however, it yields also massively increased WA equal to LSM merge operations and entail no benefits in caching behavior. A second approach is to store the physical reference of the referenced version record by a 16-byte *record id* (leaf page and slot) or only an 8-byte *page id* and successive binary search. Since covered partitions are already immutable and persistent, this might be a sound solution. Referenced version records are directly accessible, however, physical cross-references are neither desirable nor necessary in MV-PBT as outlined in Section 4.3.3. Logically referencing the comprising partition by a 2-byte partition number is more desirable. Thereby, it is possible to restore the partitioned search key and the version record is accessible with the complexity of a single cheap traversal operation. Combined effects of prefix, *Invalidation Record Value* and *Transaction Timestamp* truncation and very small values yield dense and cache-efficient key-sorted views on a subset of fragmented version records.

Creation of CP. CP are cyclically created by background processes[1]. Frequencies are possibly based on heuristics, for instance in Figure 4.22, a CP is created when 20 small partitions with fragmented version records are persisted. Furthermore, CP might comprise a subset of contents of several preceding CP, whereby these are replaced and removed from the tree structure. Whenever a CP is up to be created, the partition switch operation (outlined in Section 4.2.2) increment the most recent partition number (`max_pnr`) in the PBT cached meta data by 2 to leave space for the CP. For instance, in Figure

[1]MV-PBT applies the capability of DBMS (e.g. background writer processes [Pos21]) and K/V-Stores (e.g. eviction worker processes [Mon21]) to perform time-consuming maintenance operations in background processes and isolated by internal transactions.

4.22, partition 19 is replaced as most recent partition by 21 on switching process and 20 is able to consume the internal partial index for partitions 0−19. Ongoing modifications are absorbed by partition 21 (e.g. index record Ⓑ in Figure 4.22). The designated CP 20 is neither accessible by nor visible to concurrent transaction snapshots, hence it is possible to pause and resume CP creation based on workload properties. The background process features an administrative snapshot, whereby every version record in the immutable partitions becomes visible. It performs an *online merge sort* on the subset of preceding partitions (0 − 19) and extracts the *partition number* and *search key attribute values* from the lexicographically sorted results of the online merge sort in order to form a Cached Record, which is inserted in the CP. For instance, in Figure 4.22, the online merge sort first returns the index record Ⓒ, extracts the search key attribute value ’*Maya*’ and the partition number 4 as reference to the related value and forms the Cached Record Ⓒ’ with the partition number of CP (20). Cached Records are specifically inserted in the CP. Since result sets of online merge sort are processed record by record in a lexicographical order, Cached Records are sequentially bulk inserted in the dense-packed CP – similar to the dense-packing approach in traditional B^+-Tree structures in Section 4.2.2. The append-based behavior allows CP to sequentially evict inherent immutable leaf nodes in order keep the memory footprint and interferences with concurrent operations low. Auxiliary filter structures are incidentally created. Finally, the comprehensive and reliable CP becomes visible for querying transactions by atomically switching a flag and committing the background process transaction.

Querying CP. When background processes finished the creation of a CP, it is appropriated for querying. Covered preceding partitions are protected by the Cached Partition – i.e. they are traversed as needed without generally probing auxiliary filter structures. CP are included in the regular equality or range search process, e.g. in the online merge sort approach. Thereby, they control access to covered preceding partitions and related internally managed cursors – i.e. when a Cached Record matches the query predicates, its stored record value (the partition number of the most recent version record in the covered partitions) is used to form the search key for the specific version record in the covered partition. An internally managed cursor is applied to traverse and search for this specific version record, which is checked for visibility to the transaction snapshot. '*Visible*' version records are simply returned, however, '*invisible*' version records cause the regular search succession in the covered partitions. The latter case is very uncommon in transaction processing, since CP are created on already committed data, before the querying transaction started.

Query Example. In the example in Figure 4.22, for instance a range search operation with the online merge sort approach is performed (Section 4.2.3). The query predicates include '*Mati*', '*Maya*' and '*Meggy*'. After probing potentially available auxiliary filter structures, an online merge sort approach creates two internally managed cursors for the most recent partition 21 and the CP 20, since the other partitions are covered by CP. The first cursor (C_{M1}) traverses partition 21 from root Ⓐ and is positioned at the Regular Record Ⓑ. The second one (C_{M2}) traverses partition 20 from root Ⓐ and is positioned at Cached Record (C'). Since '*Mati*' lexicographically precedes '*Maya*', version record Ⓑ can be returned without further overhead. Subsequently, C_{M1} is moved and closed. In the online merge sort operation, (C') of cursor C_{M2} should be returned next. However, C_{M2} operates on a CP, hence (C')'s referenced version record Ⓒ is required. Partition number '*P4*' is gathered from the Cached Record value of (C') and a search key '*4 Maya*' is formed. An internally managed cursor C_{P4} is opened to exclusively traverse partiton 4 form root Ⓐ and positioned at the referenced version record Ⓒ by an equality search. The value of Ⓒ can be returned and C_{P4} is closed. C_{M2} is moved to (D'), which is within the search predicates. Similarly, cursor C_{P9} is opened to equality search for the reconstructed search key '*9 Meggy*' from (D'). Partition 9 is traversed from root Ⓐ to version record Ⓓ, which can be returned. C_{P9} is closed and C_{M2} is moved and closed, since no further records are within the search predicates. *Provided sorted view of CPs reduce* *online merge sort costs (auxiliary filter probes, cursor comparisons) with usual result sets.*

Experimental Evaluation. Effects on query performance of CP in fragmented structures are experimentally evaluated in multiple YCSB read only workloads after inserting 500k records respectively. CP are cyclically created every 20 partitions and yield an absolute number of approximately 80 partitions after 10m insertions. Figure 4.23 depicts B$^+$-Tree as non-fragmented as well as MV-PBT (w/ and w/o CP) and LSM-Trees (w/ and w/o component merges) as horizontally partitioned storage structures.

The aspired baseline is represented by non-fragmented B$^+$-Trees (gray line). They enable very stable read-only query performance, since inserted records are accessible by one traversal operation. Contrary, LSM-Trees without merging (orange line) become very fragmented by the increasing dataset, whereas its read-only throughput rapidly decrease. *MV-PBT without CP (bright blue line) suffer from similar fragmentation, however, increased cache probability of inner nodes (compare Figure 4.16) yield slightly decreasing read-only throughput.*

Approaches of regularly instructed LSM-Trees (red line) try to restore a defragmented layout by merging components with the side effect of increased WA, nevertheless, their read-only throughput is fluctuating along massively fragmented MV-PBT (bright blue line), based on their actual number of levels and components. *Although LSM-Trees (red line) spend massive effort and WA in preservation of search performance, MV-PBT (bright blue line) stay competitive without any reorganization – solely by structural benefits of a single B$^+$-Tree structure.*

Finally, MV-PBT CP (dark blue line) achieves very stable read-only performance, similar to B$^+$-Tree. *Space-efficient key-sorted view of CP retain constant performance by data skipping, when they are cyclically created at* 3, 5.5 *and* 8 *million insertions, yielding an overall read-only benefit of about* 20% *at* 10 *million insertions and a fragmented dataset of* 80 *partitions – with massively reduced WA compared to LSM-like merge approaches.*

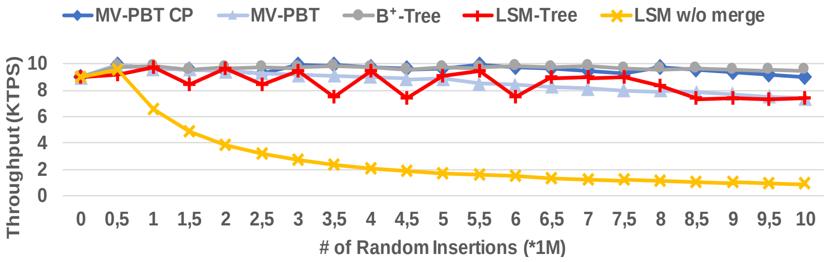


Figure 4.23.: MV-PBT with CP enable robust Query Performance in fragmented Datasets.

Cached Partitions (CP) as internal partial index are a space- and cache-efficient approach in MV-PBT to provide a comprehensive defragmented and reorganized view on data in a horizontally partitioned subset. With minimal overhead in SA and WA, read performance is comparable to defragmented layouts, whereas a robust query performance is enabled and **(H6)** *is confirmed. However, replaced and obsolete version records still increase SA, yield compelling necessity of garbage collection for endurance, nevertheless, its frequency is massively reduced by CP.*

4.4.2. Garbage Collection

In MVCC with SI, frequent modifications of logical tuples yield the existence of several tuple versions, which are forming a linked list of version records – each is valid for a different period in time across a subset of logical transaction snapshots. Whenever a version record is no more related to any active transaction snapshot, it becomes obsolete and is no longer necessary to be accessible or managed by storage and indexing structures. Version records possibly become obsolete by creation of successor version records as well as through aborts and rollbacks. Garbage Collection (GC) aims for removal of obsolete version records.

Nevertheless, obsolete version records remain in their allocated storage area for several reasons. First, especially in case of new-to-old version ordering, obsolete version records are hard to identify. Successors are neither known nor accessible by predecessors, hence obsolescence is not pretty clear. Moreover, it is not guaranteed that aborted transactions keep track about their write sets so that version records remain on rollbacks. Obsolescence must be identified by additional work and logical conclusion – e.g. by a range search operation. Second, GC require write access to comprising nodes, hence concurrency and actual payload are hampered. Third, removal of version records can cause cascading maintenance operations in order to fulfill structural constraints of storage structures. For instance, B^+-Trees leave '*ghost records*' with deletion markers behind in order to avoid node merges. Last, GC on secondary storage resident data imply WA for persistence. Version records probably relate to active transaction snapshots for long period of time, until they are replaced. Especially in case of mixed HTAP workloads, lengthy version chains to '*ancient*' version records are probably persisted until a long-lasting analytical query terminates. MV-PBT follow a consistent policy in order to remove obsolete intermediate version records, enabling space-efficient discontinuance of logical version chains.

MV-PBT GC Version Chain Discontinuance. In MVCC and SI, version records of one logical tuple form a version chain in order to provide a consistent snapshot for each active transaction. In MV-PBT, this version chain might be comprehensive, nevertheless discontinuance is permitted unless validity of transaction snapshots is violated – i.e. obsolete intermediate version records might become removed. Especially in case of long-lasting analytical queries, massive amounts of intermediate version records might are created [LSP+16]. In MV-PBT logical tuple version information is entirely materialized in each version record – except its potential invalidation. Whilst potentially succeeding invalidations of version records in the logically maintained version chains (Section 4.3.3) are determinable by querying operations, discontinuance is possible. Precautions to remove version records are as follows:

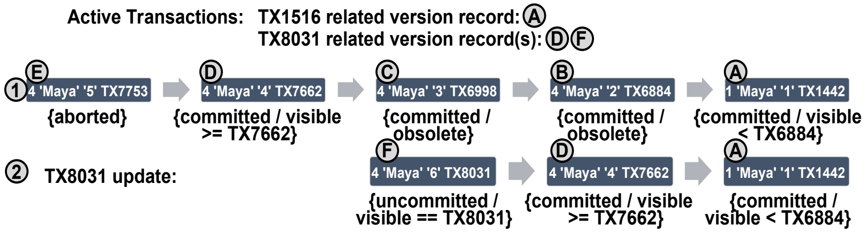


Figure 4.24.: Example MV-PBT Discontinuance of Logical Version Chain.

- creating transaction must have ended
- version record's timestamp must not be related to any active transaction snapshot
- invalidated predecessors must remain invisible; i.e. if its committed *anti matter* is removed, the *matter* itself must be also removed or invalidated otherwise (e.g. assignable successor replacement)

Within this precautions, version records belonging to aborted transactions are very easily GC'd, since they indicate the entry point or an unrelated fork. Likewise, (index-only) visibility checks (Section 4.3.6) allow to identify obsolete version records. Moreover, MV-PBT as storage management structure or index with primary key or uniqueness constraint as well as in case of logical reference indexing, every successor and predecessor version is directly assignable to every version record within the logical version chain, due to uniqueness of equal search key or virtual identifier. Thereby, several obsolete version records are eliminable without precautions. Physical reference indexing probably requires additional care and insertion of Anti Records in order to eliminate intermediate version records, since physical references vary for different tuple versions (compare Section 4.3.1).

For instance, imagine in Figure 4.24 ①, several version records (Replacement Records) of a logical tuple are created as the result of modifications and two transaction snapshots are active. An analytical query with the transaction timestamp '*TX1516*' is related to the latest committed version record before the transaction started – i.e. Ⓐ of transaction '*TX1442*'. Committed version record Ⓓ is related to the modifying transaction '*TX8031*', since its transaction timestamp '*TX7662*' is smaller than its inherent one. Other version records Ⓑ, Ⓒ and Ⓔ are obsolete, since they are replaced or their transaction did not succeed. However, '*TX8031*' is up to create a new version record Ⓕ with the value '*6*'. In ②, the forking *entry point* Ⓔ and obsolete intermediate version records Ⓑ and Ⓒ are removed, yielding a discontinuously logical version chain of version records Ⓐ, Ⓓ and Ⓕ.

Remarkable is the SA by obsolete version records, yielding cache-inefficiencies and additional search effort. Treatment of resulting WA in update-intensive workloads is challenging as well. LSM-Trees incidentally perform GC as part of excessive merging processes for defragmentation, which yield considerable WA. MV-PBT introduce Cached Partition (CP) in order to avoid effects of excessive merging. As outlined in Section 2.3.1.4, storage and index management structures should preferably perform GC for space reclamation in main-memory or at least on large segments of data, hence MV-PBT counts on miscellaneous approaches:

- Dense-Packing Phase on Partition Switch
- Cooperative In-Memory Page Level GC
- Background Online Merge Sort and Bulk Insertion

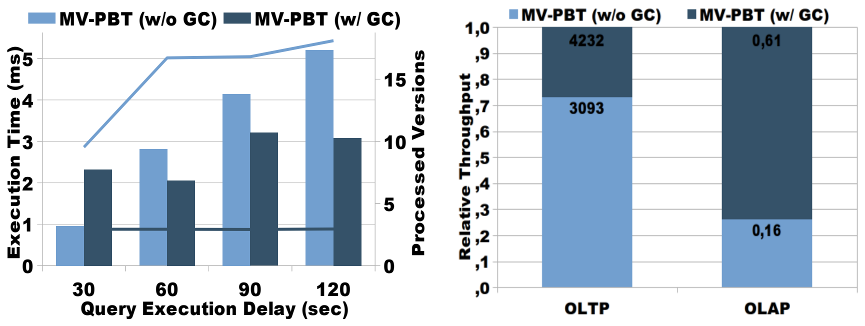
Former two approaches focus on space reclamation before data is persisted – in order to avoid WA; whereas the last approach focus on space reclamation on secondary storage devices without violating basic concepts of MV-PBT.

Dense-Packing Phase on Partition Switch. MV-PBT form immutable defragmented and dense-packed leaf nodes on partition switch (outlined in Section 4.2.2). Since leaves are subsequently persisted, obviously it is a beneficial point in time to scan the comprised dataset and finally remove obsolete version records in order to minimize SA and WA.

Cooperative In-Memory Page Level GC. Delaying GC of frequently updated logical tuples to the dense-packing phase on partition switch might incur massive interim SA and inefficient caching of version records, which are no more visible to any active transaction snapshot, yielding decreased equality and range search performance and unsaturated partitions on partition switch. Since the most recent partition is a very hot and update-intensive fraction of the tree, identification and removal of obsolete version records require special care to not impair actual payload. It must be taken into account that most obsolete version records in MV-PBT's new-to-old ordering scheme must be identified by processing the logical version chain, i.e. by performing range searches on a subset of the partition (leaves), and necessary write access for modification are mostly set on page (node) level and prevent from concurrent modification. This means that updating transactions do not recognize garbage without additional effort and GC of large fractions by range search operations or background processes shrink concurrent modifications by massive and escalating page level locking and tree maintenance operations.

MV-PBT applies a cooperative in-memory page level GC approach, whereby range scanning transactions or background processes identify and flag leaves with obsolete version records as part of the visibility check (compare Algorithm 4.5 line 14) without additional effort in the payload. Modifying transactions acquire write access in order to insert version records anyways. Moreover, they notice nodes, which are flagged for GC and initially search for and remove obsolete version records in order to reclaim space. If a node is not flagged for GC, this step is skipped for robust update performance. Finally, the new version record is inserted. Space reclamation by updating transactions leverage write accesses to nodes, reduce maintenance operation effort and operate on a recent snapshot.

For instance, in the Example in Figure 4.24 ①, while searching for its related version record Ⓐ, the analytical query of transaction '*TX1516*' identifies the version record Ⓔ of the aborted transaction '*TX7753*' and flags the comprising node for GC. In this case, the obsolete version records Ⓑ and Ⓒ are probably not identified by the analytical query, due to its '*ancient*'



(a) HTAP Query Execution Time and Version Chain Length over Runtime.

(b) HTAP Performance Gain of In-Memory GC by Transaction Processing Type.

Figure 4.25.: Effects of In-Memory GC Approaches in HTAP Workloads.

snapshot, however, they are often co-located, due to in-partition ordering conventions and lexicographical sort order. A background process would be able to operate on an administrative snapshot and identify each obsolete version record. Nevertheless it is not necessary to identify every obsolete version record by the range scanning transaction. The modifying transaction's snapshot '*TX8031*' knows recently finished transactions and recognizes version records Ⓑ and Ⓒ also to be obsolete, when searching for garbage on the write accessed node. However, version record Ⓓ is visible to its transaction snapshot and remains on the node. Moreover, it provides necessary invalidation of version record Ⓐ. Obsolete version records Ⓔ, Ⓒ and Ⓑ are removed and a new version record Ⓕ is inserted in the reclaimed space within one write access to the memory mapped node.

Experimental Evaluation of In-Memory GC Approach. Effects of MV-PBT with GC have been experimentally evaluated in an HTAP (CH-Benchmark [CFG+11]) as well as in a OLTP scenario (TPC-C [TPC10]). Both experiments are performed in *PostgreSQL with SIAS* configured with a buffer size of 600MB. Former experiments operate on a dataset size of 200 warehouses and are depicted in Figure 4.25. First, a single analytical query on '*district*' is evaluated on the system under load in Figure 4.25a. A transaction has been started and respectively idled by `pg_sleep` for 30, 60, 90 and 120 seconds in order to create several successor versions as part of the equal background load. Whilst MV-PBT without GC performs well for iterating and visibility checking short numbers of successor version records, its query execution time rapidly increase when several successor versions are created and amplify SA, WA as well as RA due to fragmentation by required partitions to consume version records. In-memory GC allow MV-PBT to identify and remove intermediate obsolete tuple versions, yielding an average of 3 accessed version records per query – i.e. a discontinuing version chain consisting of the entry point, its replaced predecessor and the visible version record. GC initially cause higher query execution time, due to increased pressure and effort on removal, nevertheless, it enables more robust query performance by lower SA, WA and RA. In Figure 4.25b, overall performance gain of in-memory GC is evaluated for the transaction types OLTP and OLAP in the HTAP scenario. OLTP throughput also increase, due to lower WA and fragmentation by 37% from 3093 to 4232 transactions per minute, however, benefits from discontinued version chains are low, since their related version record are close to the entry point in a new-to-old version ordering. OLAP operations profit from this technique and enable a 3.8 times increased throughput from 0.16 to 0.61 transactions per minute.

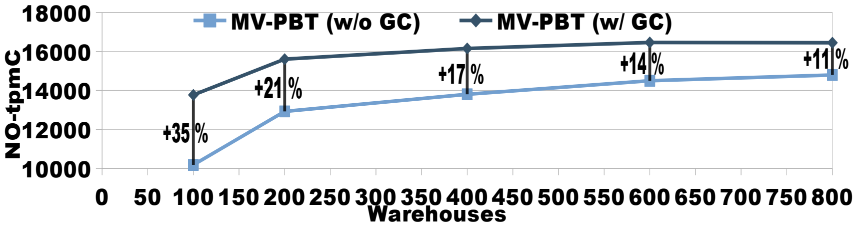


Figure 4.26.: In-Memory GC Approaches increase Throughput (depicted in New-Order-tpmC / NO-tpmC of the TPC-C workload) in OLTP Workloads. Improvements depend on the number of removable obsolete records.

In Figure 4.26, transactional throughput (in NO-tpmC [TPC10] is depicted for different dataset sizes in an OLTP TPC-C workload. In-memory GC enable robust throughput alongside all dataset sizes (warehouses), since obsolete version records are removed, yielding up to 35% performance gain. Without GC, MV-PBT gradually approach them for higher number of warehouses, since modifications spread over a larger dataset, yielding fewer version records per logical tuple – which could have been already persisted until they become obsolete. *Consequently, in-memory GC massively reduce SA, WA and RA in high concurrency situations, nevertheless, immutable and persistent partitions comprise increasing numbers of obsolete version records.*

Background Online Creation of defragmented and consolidated GC Partitions. Horizontal partitioning with immutable persistent partitions yield SA, cache inefficiency and additional processing costs, since subsequently replaced predecessor version records saturate in immutable partitions and require to be checked for visibility. Moreover, auxiliary filter structures claim caching and processing costs in order to perform data skipping on horizontally partitioned MV-PBT. In order to straighten out SA of obsolete version records, auxiliary filter structures and fragmentation, GC is occasionally performed on data in immutable persistent partitions instead of going ahead with the creation of CP. This GC approach aims on leveraging properties of MV-PBT as well as characteristics of modern storage technologies. *Flash* performs GC by copying still valid pages in new blocks and erasing their original blocks (compare Section 2.1). MV-PBT leverages this approach, however, for different units – i.e. still valid version records are copied in a not yet accessible partition by bulk insertion of the online merge sort result of a background process[1]. Since the consolidated GC partition is only accessible for the background process's internal transaction, this process is throttling and continuing without wasting work, based on payload pressure. Similar to CP, a fragmented consolidating subset of partitions facilitate

[1]MV-PBT applies the capability of DBMS (e.g. background writer processes [Pos21]) and K/V-Stores (e.g. eviction worker processes [Mon21]) to perform time-consuming maintenance operations in background processes and isolated by internal transactions.

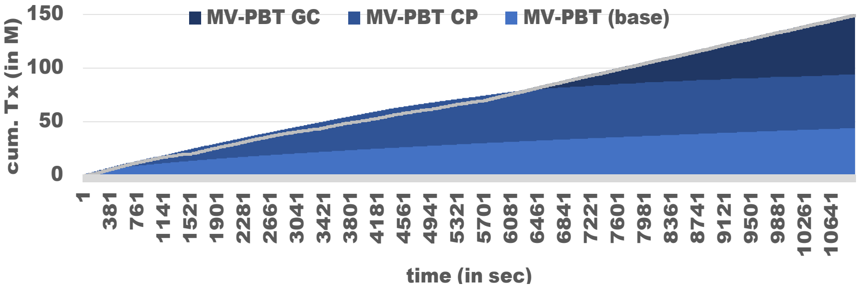


Figure 4.27.: YCSB cumulated Throughput.

ongoing query processing. Background GC processes profit from presorted views on fragmented data, i.e. the CP, since they improve online merge sort properties. In the background transaction, *matter* and *anti matter* of logical tuple versions mutually eliminate, based on the rules of *MV-PBT GC Version Chain Discontinuance*. Certainly, consolidated GC partitions aim for high proportion of *matter* – i.e. intermediate version records are removed if possible and bulk inserted records mainly contain the validation value of the most recent version record of a logical tuple in the consolidating partitions. When the background transaction succeeds, the consolidated partition becomes accessible and replaces consolidating partitions for querying. Once every active concurrent search operation terminates, the entirely replaced and fragmented subset of partitions, as well as auxiliary filter structures are allowed to be removed. Finally, apart from actual dataset and current workload, the subset of replaced and fragmented partitions is simply cropped from the tree structure by an efficient range truncation of nodes [Mon21].

Experimental Evaluation of GC Partitions. In Figure 4.27, an experimental evaluation is depicted in a write-heavy YCSB Workload A (respectively 50% update and read). Initially, the dataset size is approximately 50GB and the buffer cache is set to 100MB with 20% MV-PBT-Buffer share. Queries are executed for 3 hours and are cumulatively depicted in million transactions. MV-PBT (base/CP/GC) are colored in (bright/medium/dark) blue. MV-PBT (base) without CP and GC achieves good throughput at the beginning of the benchmark, however, it degenerates rapidly due to increasing fragmentation up to 650 partitions by 44 million transactions in 3 hours, since read I/O is dominating. MV-PBT CP achieves a much higher throughput with the few effort of cyclical internal index partition creation. Nevertheless, transactional throughput degenerates to an absolute of 94 million transactions within 1340 partitions, due to following reasons. First, CPs begin to become fragmented and would require consolidated CPs. Second, auxiliary filter structures achieve a considerable memory footprint, whereas less inner nodes in the MV-PBT are buffered in cache. Last, less caching probability of inner nodes yield slower traversal operations, due to successive read I/O, whereas beneficial effects of partial internal indexing decrease. Since updates yield cache inefficiencies and SA by the creation of version records, GC instead of consolidated CP is an appropriate approach. MV-PBT GC leverages CP to enable massive fragmentation, whereas GC becomes more valuable for the enlarged set of version records (20 CP comprising 400 regular partitions). Approximately after 20 minutes (1200 seconds), 400 partitions are generated and MV-PBT GC starts performing online merge sort and bulk insertion in a consolidated GC partition, yielding slightly degenerated throughput (hidden actual MV-PBT GC throughput is highlighted by the gray line). Nevertheless, additional effort yield robust performance characteristics and an improved cumulative throughput after 6250 seconds. Finally, MV-PBT GC processed 151 million transactions, since fragmentation is reduced from approximately 2250 to 4 completely consolidated GC partitions and residual 650 CP and CP-covered regular partitions.

In-Memory GC can minimize obsolete version records and make write I/O more valuable – i.e. SA and WA are reduced. Sparsely created consolidated GC partitions enable GC with minimal impact on payload, whereas obsolete partitions are beneficially truncated and cropped from the tree structure. **(H7)** *is confirmed. With CP and GC, MV-PBT provides robust performance without violating its principles in the horizontally partitioned structure. Nevertheless, these approaches require an order of fragmentation to take effect, whereas* *auxiliary filter structures* (compare Section 4.4.3 and Chapter 5) *reliably exclude unrelated partitions, when they are created.*

4.4.3. Application of Auxiliary Filter Structures and Data Skipping

(MV-)PBT is an evolving horizontally partitioned tree structure. Result sets require to consider contents of each partition. In (MV-)PBT, this might happen by root to leaf traversals in every partition. A moderate number of required traversal operations is well covered by commonly cached and utilized inner nodes as well as asymmetry and parallelism in modern storage devices (Flash), however, processing and read I/O costs accumulate and probably yield inadequate query performance, even if the result sets of most partitions are empty.

Insufficient Cached Partitions. Data skipping approaches in (MV-)PBT facilitate to avoid traversal of partitions without related records. Cached Partitions (CP) are already introduced in Section 4.4.1 as very accurate and incorporating data skipping method, nevertheless, they *require an order of fragmentation to take effect* and build a lexicographically sorted view on a subset of preceding partitions.

Characteristic Prerequisites. Auxiliary filter structures are already introduced in Section 4.2.1.1. They constitute practicable approach for membership testing on query predicates in each partition, before traversal operations are performed. Nevertheless, entire datasets are not fully represented, whereas results are probably erroneous. Auxiliary filter structures are only applicable for data skipping in (MV-)PBT, if errors probably yield additional work, but ensure integrity. Hence, they must guarantee absence of version records with the specific query predicates or lead to traversal operations. For this reason, results of auxiliary filter structures are *negative*, *true positive* or *false positive*. The ratio of *false positives* and absolute filter requests is the *false positive rate (FPR)*, which is intended to be optimized. Only in case of *negatives* traversal operations are saved, otherwise the effects are only *additional probing costs*.

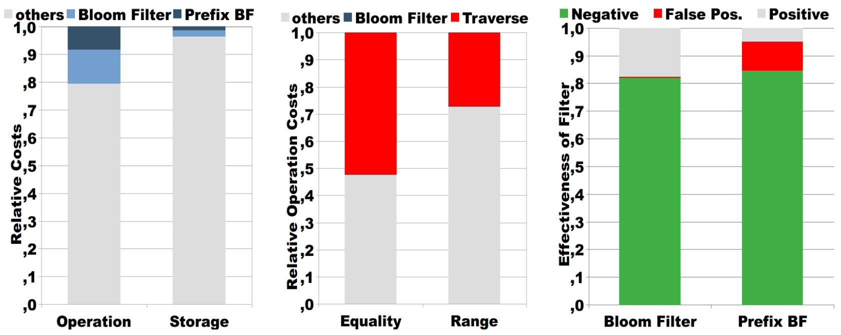
Cost Trade-Offs. Probing auxiliary filter structures accompany processing and memory costs, which require to be taken into account [LNKB19]. Different approaches vary in computing complexity. Crucial factor in overall performance and costs stem from additional memory costs, which are a kind of SA. Considerable filter probing effort require to be significantly lower than actual costs. In MV-PBT, for instance, commonly cached inner nodes take high contribution in overall performance. Auxiliary filter structures might cause inner nodes to be evicted from database buffer and yield worse performance characteristics. Hence, auxiliary filter structures should have well performing *caching characteristics* and a *low memory footprint* beside manageable *processing effort*.

Auxiliary filter structures are classified in very *lightweight online maintainable but inaccurate filter* as well as very accurate structures for *approximate membership testing*, which are asynchronously generated on partition switch (Section 4.2.2), since online maintenance would adversely influence throughput of payloads.

Lightweight Online Maintainable Filters. These auxiliary filter structures are very cheap to probe and maintain, however, they are rather a rough digest of protected dataset. In MV-PBT, exclusion by these structures happens by probing and logical conclusion on mutable as well as immutable partitions. They are possibly disqualified by mismatching search key values or temporal dimension gathered from query predicates and transaction snapshot. *Partition Fence Keys* are applied on search key query predicates. They are a memory representation (Cached Meta Data in Section 4.2.1.1) minimum and maximum search key value of comprised version records in a partition. Probing and maintaining partition fence keys is very cheap, however, facilitate data skipping only if probed search keys are not within a partition's represented range, whereas its benefits are limited. *Minimum Transaction Timestamps* reveal from specific partitioning characteristics of MV-PBT and indicate the lowest transaction timestamp of comprised version records within a partition. If it is larger than any concurrent transaction timestamp of a snapshot, visibility checks would not identify any comprised version record to be visible, hence traversal in its partition can be saved (compare HTAP query characteristics in Section 4.3.5). Especially recently created partitions can be skipped. Moreover, long-lasting analytical queries probably are able to skip multiple recently created partitions. However, this lightweight filtering technique is only applicable for succeeding partitions, which follow the transaction snapshot.

Structures for Approximate Membership Testing. These kind of auxiliary filter structures form a very accurate but more expensive group of data skipping approaches. Appropriate operational purposes of commonly available filters are probing query predicate search key attribute values for *arbitrary equality searches* as well as *specific range search operations*. Well-known representatives are (prefix) bloom filters and derivatives [AK21; BM03; LGM+18; TRL12]. Calculations and space requirements for probing and maintenance are manageable and significantly lower than actual costs of root to leaf traversals to unrelated partition contents. Thereby, maintenance is beneficially performed apart from payload as part of the partition switching operation (outlined in Section 4.2.2), since its final immutable regular search key-set is known and can be cache-efficiently bulk inserted in a well-sized filter with low impact on payloads. When the asynchronously created filter(s) are finalized, they become available for querying by atomically setting a flag in the partition meta data (Section 4.2.1.1). Query predicates that do not represent comprised version records' search key attribute values within the filter cause a search operation to skip the related partition.

Experimental Evaluation. Effects of data skipping by bloom filters for equality searches as well as prefix bloom filters on a customized set of search



(a) Creation and Storage Costs. (b) Supported Operation Costs. (c) Negative Results save Traversals.

Figure 4.28.: Dimensions of Approximate Membership Testing in MV-PBT.

key attributes for range searches are experimentally evaluated for (MV-)PBT in *PostgreSQL with SIAS*. A TPC-C workload is instrumented to operate on a large dataset (2000 warehouses) and a total database buffer size of 600MB (inclusive 20% MV-PBT-Buffer). Reported results are averages across given transaction processing and related objects in the database schema, except where otherwise specified.

First, costs of (prefix) bloom filter maintenance are evaluated and depicted in Figure 4.28a. Approximately 20% of relative operational costs (in clock cycles) of the entire partition switching operation (outlined in Section 4.2.2) arise on filter creation (brighter and darker blue). It is a significant proportion of the total partition switching process, however, bloom filters are asynchronously created in the background, hence operation costs have low effect on processing of actual payload. In contrast to operational effort, storage costs are relative low. Regarding the total size of partition leaf nodes, (prefix) bloom filters occupy relative low memory (less than 4% in total), since only search key attribute values are inserted in a lossy compressed filter structure. This is possible, since knowledge of record values or reconstruction of materialized search key attribute values is not facilitated in auxiliary filter structures.

In Figure 4.28b, relative processing costs (in clock cycles) of traversal,

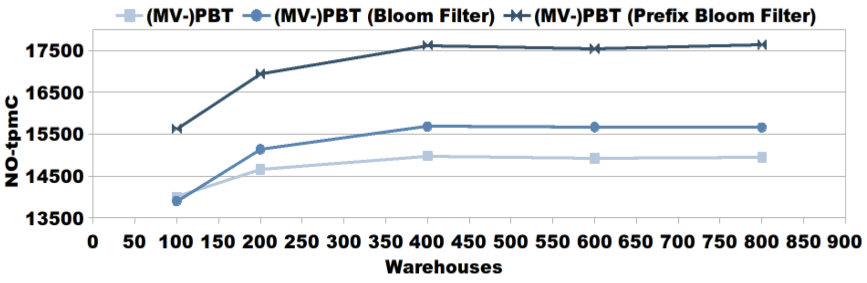


Figure 4.29.: TPC-C Throughput (depicted in New-Order-tpmC / NO-tpmC) for MV-PBT with unprotected, bloom filter-applied as well as in addition prefix bloom filter-protected immutable Partitions.

filter probe and residual index operations as well as tuple retrieval costs (others) for equality and range search are depicted. Traversal operations are very processing-intensive with noteworthy memory footprint of cached inner nodes (which are otherwise required to be fetched from secondary storage devices) of the traversal path. Other major proportion includes visibility checks and tuple reconstruction costs from base table, which require to be performed anyways. Probing bloom filters incur less than 0.3% of total operational costs with a comparably low memory footprint. In case of range searches, proportion of other operations increase, since one traversal allow to iterate several version records of different logical tuples. Nevertheless, traversal operations are a major proportion of query operations, which need to be avoided for unrelated partitions.

In Figure 4.28c it is shown that for this workload settings (prefix) bloom filters are able to save about 81.6% of traversals for equality and 84.5% for range search operations. Only 0.6% false positives cause traversals in unrelated partitions for equality search operations. Nevertheless, insufficiency of highly customized prefix bloom filters yield 10.6% false positives.

Effects of approximate membership testing on throughput (TPC-C NO-tpmC) for several dataset sizes are depicted in Figure 4.29. Bloom filters enable the equality search to save traversals and accompanying database buffer evictions as well as read I/O, at best yield an overall benefit of 5% in throughput. *Additional customized prefix bloom filters* have a similar effect on very expensive range search operations and achieve an additional benefit of approximately 12.5%.

Auxiliary filter structures in (MV-)PBT enable data skipping of unrelated partitions. Processing costs, database buffer hit characteristics and accompanying read I/O are cheaply optimized in horizontally partitioned storage and index management structures yield significantly improved performance characteristics – hence **(H8)** *is confirmed. Nevertheless, data skipping rely on separately created and cached filter structures for equality and customized range search operations, even though identical datasets are covered. Moreover, prefix bloom filters are indeed capable to perform approximate membership testing on specific range search predicates, but are impractical for arbitrary search predicates in MV-PBT and result in high FPR* (compare Figure 4.28c).

4.4.4. Summary

(MV-)PBT natively leverage their inherent structural properties in order to adaptively reorganize horizontally partitioned view on comprised data and retain beneficial search performance characteristics of defragmented B^+-Tree structures (compare Section 3.1.1). Thereby, SA as well as WA is kept at a low level to minimize RA by append-based and sequential bulk insertion (Sections 4.4.1 and 4.4.2) of version records (Section 4.3.1). A discontinuously logically maintained version chain (Section 4.3.3) allow (MV-)PBT to perform GC (**(H7)**) as well as data skipping and key-sorted views by CP (**(H6)**). Moreover, auxiliary filter structures (Section 4.4.3) are created upon record's search key attribute values for each partition in order to cheaply enable general data skipping (**(H8)**). (MV-)PBT facilitate robust performance characteristics by inherent techniques (Figures 4.23, 4.25a and 4.26), however, respective filter structures incur redundancies (Figure 4.28a) for equality and range search as well as increased FPR for arbitrary custom query predicates in MV-PBT.

4.5. Conclusion

In this chapter, PBT has been introduced as temporally evolving horizontally partitioned storage and index management structure which is leveraging characteristics of flash on secondary storage technologies. Fields of application as well as maintenance and compression techniques (**RQ3**) are inherited from the structure of a B^+-Tree, nevertheless, (MV-)PBT achieves benefits in SA and buffer efficiency. High performant maintenance of appended circumstances in PBT accompanying well with new-to-old ordering of physically materialized version records with logical out-of-place update scheme and one-point invalidation model. Annotation of transaction timestamps to each version record enable MV-PBT to perform (index-only) visibility checks as part of the regular search operation, whereas **RQ1** is answered. Moreover, **RQ2** is addressed by appropriation of MV-PBT as multi-version storage management structure, which beneficially identifies rolling new-to-old version chain entry points of logical tuples by searching their primary keys. Nevertheless, fields of application and integration are further elaborated in Chapter 6. GC is performed in main-memory or on asynchronous background bulk merges of massive number of partitions (**RQ5**) yielding minimal WA and sequential writes for space reclamation. This and adaptive reorganization of read optimized views by CP (**RQ4**) is possible, due to the discontinuance of logical version chain property, whereby reorganizations in dedicated partitions are possible. Moreover, CP as well as auxiliary filter structures enable data skipping (**RQ3**), whereas traversal operations and read I/O is saved.

It is shown that MV-PBT leverages secondary storage technology characteristics in a tree-based structure and addresses rising challenges in the same way coping with modern update-intensive mixed workloads. Hence, MV-PBT is qualified for system integration in Chapter 6. Nevertheless, popular existing auxiliary filter structures for data skipping require high administrative effort and redundancy for arbitrary querying. For this reason, an inclusive and space efficient point-range filter approach for arbitrary approximate membership testing is introduced in Chapter 5.

Chapter 5

Arbitrary Approximate Membership Testing

Horizontally partitioned storage and index management structures, like LSM-Trees or MV-PBT, maintain (at least temporarily) semi-sorted and fragmented datasets. Equality and range searches potentially require to traverse every fragment from its root to leaves in order to build reliable result sets. Based on the selectivity of query predicates and fragmented contents, several relatively expensive traversals yield empty result sets. Therefore, significant processing, caching and read I/O costs occur.

Auxiliary filter structures enable approximate membership testing, whereas a negative result is guaranteed to be correct. A protected fragment by such a filter structure can be readily skipped on negative filter probe when building result sets for equality or range search operations. However, existing approaches are costly and inadequate for arbitrary membership testing.

Point-range filters (PRF) are cost-benefit-optimal auxiliary filter structures for arbitrary membership testing. They fully enable data skipping in partitioned storage and index management structures raised by (**RQ3**).

5.1. Problem Statement

MV-PBT is a horizontally partitioned storage and index management structure. Search operations principally require to consider each partition to build reliable result sets. MV-PBT protects each immutable partition (Section 4.2.2) by *auxiliary filter structures* (Section 4.4.3), which are probed first before costly traversal operations are performed on search operations (Sections 4.2.3 and 4.3.5).

Section 4.4.3 demonstrates the significance of *auxiliary filter structures* in equality and range search operations in MV-PBT. For instance, under the configured TPC-C workload characteristics, more than 80% of partitions are potentially skipped as a result of a negative filter probe (compare Figure 4.28c), with the result of significant performance gains (compare Figure 4.29). Especially, costs of range searches in MV-PBT are reduced by appropriate auxiliary filter structures, since there is no inherent breaking condition in the range search algorithm.

Nevertheless there is a major problem with the applied approach. Whilst point filter sufficiently enable arbitrary membership testing, benefits of additionally maintained and highly customized prefix filters are limited at high costs. Probing arbitrarily selectable range intervals in typical query predicates causes amplified costs for individually maintained prefix filters or reduces accuracy, due to inappropriate filter configurations. Either way, it might be more profitable to save memory footprints of prefix filters and increase cache efficiency of inner nodes in MV-PBT (compare Section 4.2.4, especially $p_{ic_{PBT}}$ in Equation 4.3). *Multiple prefix filters are unacceptable, due to their adverse cost-benefit ratio. Especially, it should be noted that required information is present in point filters – however, impracticable for probing. The proposed approach, bloomRF, is designed as unified efficient point-range filter (PRF) for arbitrary approximate membership testing and an appropriate substitution of specialized bloom filter approaches (BF).*

5.2. Point Filter Techniques

Bloom filters (BF) are the most popular approach for approximate membership testing. A variety of derivatives [AK21; BM03; LGM+18; TRL12] aim to cover different scopes like online maintainable entry sets by counting [BMP+06; FCAB00; RKK12] or other partially deletable encodings [RMVM10]; as well as cost performance trade-offs regarding hash efficiency [DM04; KM06], compressibility [Mit02], data locality [CMB+10; DSL+11; LDD11; LNKB19; PSS10] and vectorized processing [LNKB19; PR14] on modern hardware. BF and other point filters, like Cuckoo filter [FAKM14], with different properties have been proposed in numerous novel publications [DAI18b; DT21; GYC+21; PCD+21; TC21] including learned approaches [KBC+18; Mit18] for interesting applications [DHI20; LZSC20; VKKM20] and are already state-of-the-art in high performance K/V-Stores [GD22; Inc22; Mon21].

General View on Bloom Filters (BF) and Derivatives. Generally, BF store a space efficient hash-generated encoding of k bit positions for each inserted element in an initially zero allocated bit array of length m by flipping the bits at calculated positions to 1. While inserting a set of n elements, several positions are affected one or multiple times, whereas others are never attained and remain 0. A query operation utilizes equal hash functions to lookup at k encoding bit positions of arbitrary elements. Results of approximate membership testing in BF return a negative result for lookup of arbitrary elements, if at least one of k calculated bit positions is not set to 1, since every k bit positions of contained elements must be set. Hence, approximate membership testing return a positive result if every k bit positions are set. This could also be the case if all k bit positions of a probed element are encoded by other elements of the comprised entry set, with the result of a false positive. The ratio of false positives and absolute filter requests is the false positive rate (FPR). By varying the filter size m for an entry set of n elements and consequently altered number of encoded positions k on creation, an acceptable probabilistic FPR for certain costs is configurable

[Blo70].

Cuckoo Filter. Cuckoo Filters [FAKM14] are introduced as ’deletable’ and ’practically better’ BF alternative. Short hashed signature tags (fingerprints) of an element are inserted in a hash table at one of several possible locations in order to handle collisions, however, insertion errors might occur. Probing an element requires to calculate its fingerprint and to lookup possible map locations, what result in a constant number of cache misses and moderate FPR. Nevertheless, BF overtake Cuckoo Filters with regards to cost-benefit trade-offs [LNKB19] for different applications.

BF in Partitioned DBMS and K/V-Stores. In temporally evolving horizontally partitioned structures, assigned point filters probably protect fragments from being accessed, if probed elements (search key attribute values) are not included. Assuming an equal hash function for each assigned filter, probing an element entail equal positive integer hash values, which are aligned to appropriate positions in different sized filters by a modulo m. Hence, processing effort is nearly constant for several probed filters. Moreover, BF can reduce processing costs by leveraging double hashing techniques, whereas positions are not independently calculated with negligible impact on FPR [DM04; Mon21]. Several BF approaches focus on restructuring access patterns in order to achieve performance effects by data locality [CMB+10; DSL+11; LDD11; LNKB19; PSS10]. Finally, overheads require to be considered in two dimensions for different tasks in DBMS and K/V-Stores – i.e. additional work on false positives as well as memory occupancy and processing costs by the filter structure [LNKB19]. Hence, following observations are deduced from recent BF approaches:

- reusability of hashes in multiple BF-instances per probed element
- relation in calculated positions is conceivable for independently generated hash values
- restructuring enables data locality for various objectives

- considering cost per farther-reaching benefits in performance optimal filtering

BFs are efficient and compact probabilistic structures for approximate membership testing, which guarantee configurable cost-performance trade-offs. Processing costs are nearly constant with a linearly growing memory footprint per horizontally partitioned fragment of a dataset, whereas they are suitable for a moderate number of probed partitions in MV-PBT. Nevertheless, benefits per cost are limited for arbitrary membership testing, due to their hash-dependent limitation in applicability, since every bit position considers the entire element (search key attribute values) for exact match filtering.

5.3. Broadened Range Filter Techniques

Point filter techniques received high attention in the past, however, their capabilities and application are limited to exact match filtering. Horizontally partitioned storage and index management structures, like LSM-Trees or MV-PBT, also support range search conditions, which are very likely in OLTP or HTAP workloads. Hence, these requirements must be considered by auxiliary filter structures for data skipping in MV-PBT in order to achieve farther-reaching benefits per costs for performance optimal filtering.

Challenges in Range Filtering. BF encode elements by setting hash-generated bit positions. Therefore, the entire element, e.g. search key attribute values, are appropriated to generate a randomly spread encoding in the BF. In order to probe an arbitrary range of elements, e.g. integers between 45 and 60, $15 \times k$ randomly spread cache-inefficient probes are necessary to probably return a negative result. Cuckoo Filters are subjecting similar complexity rules. Moreover, error rate (i.e. FPR) in filter structures make a negative result with increasing range predicates unlikely. For instance, already small range probes on floating points, like elements between $1.49d$ and $1.51d$, incur millions of probed elements. Complexity also increases by multiple search key attributes. Moreover, in non-scalar datatypes, the set of probing elements is infinite. Hence, a point-range filter requires to support:

- arbitrary element distribution and range spans
- common scalar and non-scalar datatypes in DBMS
- multi attribute search key values

Prefix Bloom Filters (PBF). PBF encode positions and probe customized-length prefixes of elements. Hereby, range probes in the exact dyadic interval span of custom prefixes are enabled. Nevertheless, arbitrary membership testing is not possible, since larger range spans incur multiple probes (like in BF) and non-dyadic or smaller intervals as well as point probes suffer from information loss by the prefix, with the result of increased FPR. Arbitrary approximate membership testing would require several filter structures, with the result of poor cost-benefit ratio.

Dyadic Intervals (DI) and Interval Decomposition. Prefixes in PBF are represent to a specific DI in the domain of a datatype, e.g. a 2-byte unsigned

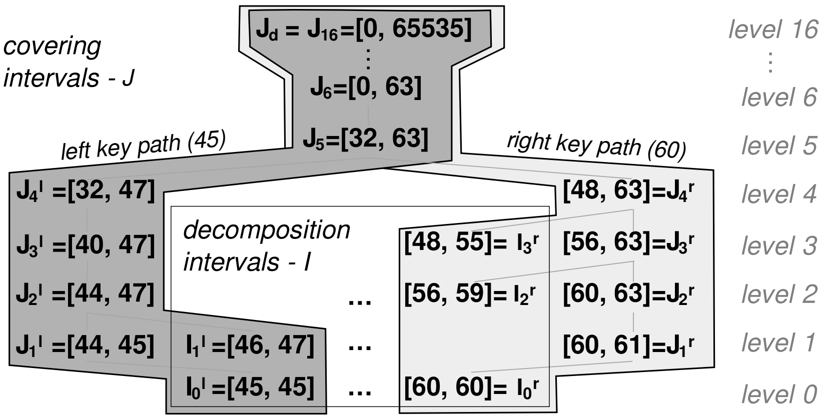


Figure 5.1.: Decomposition in Dyadic Intervals (DI) in Levels for range query I = [45, 60] in a 2-byte integer domain ($d = 16$). [MRBP23]

integer as depicted in Figure 5.1. DIs are characterized by *power of two* inclusive lower and exclusive upper boundaries, organized in an adequate number of levels to represent the domain, e.g. 17 levels for unsigned 2-byte integers (depicted as levels 0 to 16). Each level can be decomposed in *two* subordinated levels which form a binary tree, e.g. the second DI on level 5 $J_5 = [32, 63]$ can be decomposed on level 4 in $J_4^l[32, 47]$ and $J_4^r[48, 63]$. Any randomly selected interval can be decomposed in a set of DIs, serving as information in arbitrary approximate membership testing. Thereby, DIs on different levels fully cover (denoted as J), partially cover (denoted as J^l or J^r for left and right path) or are a full or partial decomposition (denoted as I^l, I^* or I^r for left, inset or right path) of a randomly selected interval (denoted as I), compare Figure 5.1.

5.3.1. Related Work

Rosetta – Probabilistic Multi-Level PBF Approach. Based on the scheme of DIs and implicit segment trees [BCKO08], Rosetta [LCK+20] encodes elements and several variable length prefixes of DIs in a respective hierarchical set of (prefix) BFs. In order to identify the necessary set of BFs and their respective variables (m, n, k, respective FPR), the set of elements as well as most frequently probed range interval span must be known at construction time, whereas an offline creation process is required. Generally, this restriction limits the applicability of Rosetta, nevertheless, its focus is on protecting unrelated immutable LSM-Tree components in K/V-Stores from being attained by equality and short range query predicates ($2^1 - 2^6$). A covering common prefix of lower and upper bound interval keys and its related DI with respective (prefix) BF are determined as root of the probing process. For instance, in Figure 5.1, the covering interval of $I = [45, 60]$ is $J_5 = [32, 63]$, i.e. the second DI in level 5. On positive result, Rosetta begins with '*doubting*' – i.e. decomposed DIs (J^l, J^r, I^l and I^r) in lower levels are probed, whereas the recursive process of negative DIs breaks. Whenever a hierarchically and recursively processed probe of related DIs return positive results up to level 0, Rosetta indicates the probe on I positive, contrary, if every related DI can be excluded at a level, Rosetta returns a negative result. Therefore, Rosetta's probing costs per interval range span even out between logarithmic and linear complexity, which probably exceeding saved I/O costs. Moreover, amplified probing effort per interval range span increase FPR [LCK+20], since each probed element is subjecting an accumulated FPR calculation. This is aggravated by out of order probing interval range spans, which a Rosetta instance is not configured for.

Considering recent trends in BFs, Rosetta impedes reusability of calculated hashes for multiple BF-instances, due to the unpredictable process of doubting and involve significant costs for a small range of inelastic applicability – whereas probing costs are quite able to exceed benefits, since several probes of course have positive results [ZCWJ21]. This might be practicable in some LSM-based K/V-Stores, however, arbitrary element distribution and range interval spans are not optimally handled in Rosetta.

Succinct Range Filter (SuRF) – Deterministic Trie Approach. A complete different approach is adopted by SuRF [ZLL+18]. This filter stores an entirely materialized element (-prefix) in fast succinct tries (FST) rather than relying on DI and implicit segment trees. Based on the data distribution and length of common prefixes, tries enable space efficient encoding schemes. SuRF relies on this techniques and truncates expensive variable length suffixes, which become replaced by a small lossy compressed representative (hash, truncated suffix or both) in an offline creation process. Since the prefix is entirely materialized, no errors occur in this range area, nevertheless, SuRF suffixes are very erroneous, based on the assigned size. Hence large range interval spans ($2^{37} - 2^{38}$) are well covered by SuRF, however, smaller intervals require to virtually materialize entire elements, amplifying the filter's memory footprint.

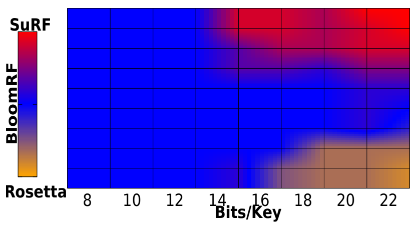
Applicability of SuRF in horizontally partitioned storage and index management structures is limited, due to significant memory costs per benefit for arbitrary elements and range interval spans. Extended element sizes in non-scalar datatypes and respectively acceptable memory footprints potentially improve benefits in SuRF, though very uncommon as search key attribute values in key-sorted structures. Reusability of calculations is not met by trie structures, nevertheless, processing costs are very cheap.

Adaptive Range Filter (ARF). An early filter structure coping with range query predicates on a predefined dataset is ARF [AKL13]. It is also based on a trie structure, however, its contents are learned during a training phase – indicating the existence or non-existence of an element in the covered dataset. Nevertheless, this filter aims for cold store protection and is not suitable for arbitrary approximate membership testing in horizontally partitioned storage and index management structures, due to its cost-benefit ratio and time consuming training phase.

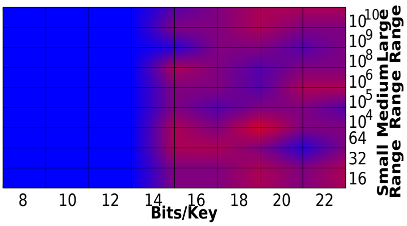
Existing approaches for approximate membership testing are not able to cover a significant fraction of the problem space for performance optimal filtering of arbitrary range interval spans by constant costs. Hence, their applicability in horizontally partitioned storage and index management structures is limited.

5.4. bloomRF: Unified Point-Range Filter for Arbitrary Approximate Membership Testing

bloomRF is designed to efficiently cover the entire problem space of arbitrary approximate membership testing with focus on benefits by broadened area of predictably valuable applicability with constant costs like BFs. In Figure 5.2a, an excerpt of best FPR for bloomRF (blue), Rosetta (yellow) and SuRF (red) is given for several range interval spans and filter sizes. Rosetta is able for specific dataset and workload distributions to achieve slightly improved FPR for very short ranges and high space requirements – e.g. 16 bits/key, what actually is a quarter of the entire element size of 64 bits. Thereby, Rosetta is not able to reach its competitor's latencies in any case, as depicted in Figure 5.2b – i.e. benefits per costs are limited. SuRF, on the other hand, covers very



(a) FPR per Filter Size and Range



(b) Latencies per Filter Size and Range

Figure 5.2.: bloomRF (blue) predominates over competitors SuRF (red) and Rosetta (yellow) in the problem space of arbitrary approximate membership testing by area of applicability (range interval span 16 to $1E+10$ – ordinate) and costs (filter size 8 to 22 bits per key – abscissa). Colors indicate the approach with the best average performance (FPR or Latencies) for various dataset sizes ($1E+3$ to $5E+7$) and normal distribution [MRBP23]

large range interval spans, whenever the dataset is sufficiently represented in the trie structure. For instance, SuRF is not able to form a trie for every dataset size and arbitrary filter sizes in bits/key, limiting its benefits per costs. Probing latencies, however, are very comparable to bloomRF, whenever SuRF becomes operable (compare Figure 5.2b, latencies are similar to the right of 14 bits/key), though significance is limited in library evaluation, due to caching effects. bloomRF exhibits constant performance in the entire problem space, even in case of lower memory footprints. Moreover, occupied areas of SuRF (red) and Rosetta (yellow) indicate by no means deficiencies of bloomRF, rather than slight improvements by focused strengths of its competitors. bloomRF takes most benefits from additional costs.

The central idea in bloomRF is to encode range information in the hash code itself. Based on the properties of DIs a concept of implicit dyadic trace trees (DTT) is introduced to encode range information by prefix hashing and piecewise monotone hash functions (PMHF) in a bit array similar to regular BFs. Principles are outlined in the following.

5.4.1. Implicit Dyadic Trace Trees (DTT)

bloomRF conceptually encodes structured range information in traces. Each element in a scalar datatype's domain is placed at its designated position in a trace, since it respectively has one directly associated smaller and larger adjacent element. By this means, each element in a domain with the size 2^b is encoded by one respective position in an equally sized trace. For instance, in a domain D_3 of 2^3, elements $[0,7]$ are encoded in a trace of size (s) 8. In order to insert the element 5 in a trace, the sixth position is set (since it is zero based) with the result '*0000 0100*' in bit representation. Existence of an element is identified by probing whether its respective position in the trace is set. In the example, element 5 is set to 1, but 4 is non-existent since its position is not set.

Traces are collapsed Binary Trees. Considering DI's binary tree nature, by insertion of an element in level 0, intervals in its respective composing levels $\{1, 2, \ldots, levels + 1\}$ must also be set. Traces logically build an implicit binary tree, named trace tree, of height $h = b + 1$, including a virtual root

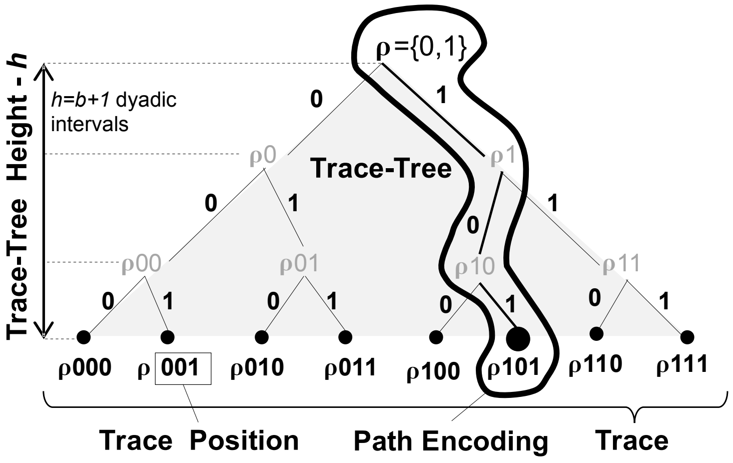


Figure 5.3.: Structure of a Trace Tree. [RBMP20]

(depicted in Figure 5.3). Following the binary tree path, every respective element position forms a binary path encoding – partially sharing common prefixes. These represent h levels of DIs. Hence, setting an element's position in a trace induces the existence of h related levels of composed intervals.

For instance, the existence of element 5 (p101) by setting the sixth position in the trace (t_s = *0000 0100*) allows to identify also its composed dyadic intervals $[0,7];[4,7];[4,5];[5,5]$ in levels $3;2;1;0$ respectively. Thereby, any arbitrary (range) query interval composed of DIs across several levels are testable by one interval trace t_i. For instance, DI $I_D = [4,5]$ is translated in the interval t_i = '*0000 1100*' and probed by a bitwise 't_s `AND` t_i `NOT =` 0' in a single trace (word) access. In addition, several decomposition DIs are composable in one t_i, e.g. arbitrary range query interval $I_A = [3,5]$ can be decomposed in $I_{A1} = [3,3]$ and $I_{A2} = [4,5]$ but also translated in t_i = '*0001 1100*'. *As a result, information of h DI levels are space efficiently represented and cheaply probed by the logic of implicit trace trees.*

Dearness in large Binary Trees. Trace trees probably vary their domain size and increase the covered DI levels. By this means, the height of trace trees is increased for each additionally comprised level, whereas necessary trace size exponentially grows (compare Table 5.1). Encoding and probing traces with exponentially growing sizes become cache and processing intensive. Hypothetically, if assuming bit representation for element positions, a continuous space of 2.3 exabyte is required to represent and probe the domain of 8-byte integers.

Reduction of Complexity in directly accessible Sub-Trees. The idea is to subdivide the entire domain in a novel structure of implicit dyadic

b	correl.	1	2	3	4	5	6	7	8	...	64
h	$b+1$	2	3	4	5	6	7	8	9	...	65
s	b^2 [bits]	1	4	8	16	32	64	128	256	...	2.3 EB

Table 5.1.: Correlation of Trace Tree Properties.

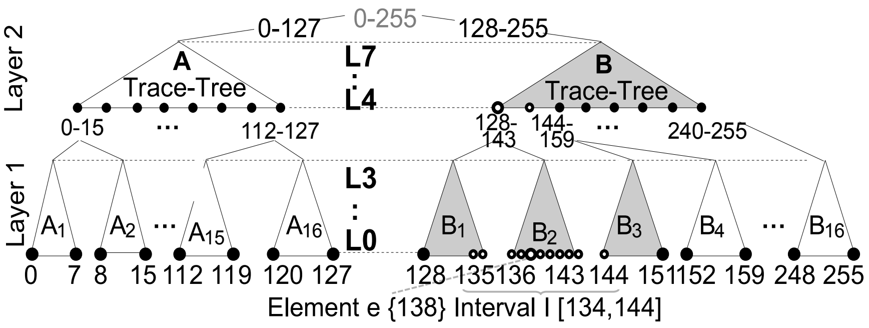


Figure 5.4.: Layers of a Dyadic Trace Tree (DTT) (compare [RBMP20]).

trace tree (DTT) nodes, since sub-trees adopt the characteristic of a tree. A balanced DTT represents the entire domain by a set of trace trees, which are individually addressable nodes of a DTT. Trace trees are arranged in hierarchical layers, which respectively cover a subset of the entire DTT height $|H|$. By traversing the tree path, root and inner nodes build a cumulative prefix of subordinated inner nodes and leaves (layer 1). Thereby, prefixes themselves are addresses of subordinated nodes, wherefore it is possible to encode levels of the binary tree as a virtual root of trace trees – i.e. each trace position in non-leaf trace trees is split in two child nodes with high density in information.

For instance, in Figure 5.4, the domain of 2^8 (1-byte character) is represented by a DTT. It consists of two layers of nodes formed by trace trees, each of them is covering a domain of 2^3 at its respective level. A logical root at dyadic level 8 is subdivided in trace trees A and B at layer 2 – representing levels 7 to 4. Furthermore, a prefix of 5 bits is represented by layer 2 and a logical root. Trace tree A represents elements $[0, 127]$ and B $[128, 255]$ as well as a set of 16 elements per trace position in level 4. Each of the 8 element positions in trace trees A and B, two nodes of layer 1 are assigned. For example, the first trace position of B covers the DI $[128, 143]$ and is divided in $B1$ and $B2$, which comprise elements of DIs $[128, 135]$ and $[136, 143]$ respectively. Within their 8 trace positions, the actual element is deduced by the trace tree address, which comprises the entire prefix.

Deterministic Arbitrary Membership Testing in DTT. Probing arbitrary points or ranges in DTT probably involves several trace trees of different layers. Each is individually testable with positive or negative results. An element is existent, if every probed position in respective traces is set, otherwise its existence is impossible.

For instance, testing for element e {138} in the DTT of Figure 5.4, involves trace trees $B2$ but also B, since the element is within B's DI of $[136, 143]$. A trace at the address of $B2$ is compared with trace position, comprising the DI of the element e, i.e. {138} at level 0 is transformed in the trace '*0010 0000*' with regards to its prefix. In B, the element 138 can be optionally verified by additionally probing the trace position '*1000 0000*'. *By this means, elements are deterministically verifiable by probing comprising DIs.*

Covering Intervals and Decompositions. In DDTs, trace positions in several layers represent DIs at specific levels. Hence, they allow probing of fully and partially covering intervals or decompositions of range query intervals. Thereby, hierarchical layers cover different prefixes and elements.

In Figure 5.4, considering a range query interval I $[134, 144]$, which is decomposed in $J_5 = [128, 159]$, represented by $t_B =$ '*1100 0000*' and probed in B. The left path is comprised in $B1$ ($[128, 135]$) which includes covering intervals of levels 4, 3 and 2 as well as full decompositions in level 1 and 0 denoted as $I_1^l = [134, 135]$ (according to notation in Figure 5.1) as a composition of $I_0^l = [134, 134]$ and $I_0^l = [135, 135]$), which are commonly represented in the trace t_{B1} ='*0000 0011*'. $B2$ comprises of full decompositions in the range $I_4^* = [136, 143]$, represented by $t_{B2} =$ '*1111 1111*', however, this optional probe is a fully included decomposition of level 5. The right path comprises $J_4^r = [144, 159]$, whereas $B4$ ($[151, 159]$) is not part of the decomposition and must not be probed. Decomposition interval $I_0^r = [144, 144]$ is represented by $t_{B3} =$ '*1000 0000*' in $B3$.

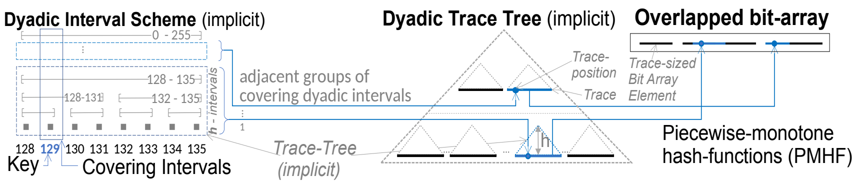
A range query interval $[134, 144]$ in the domain 2^3 is tested in DDTs by probing 3 decomposed traces (t_B, t_{B1}, t_{B3}) and can be optionally verified in t_{B2}.

Probabilistic Error Tolerance in DTT enables lossy Compression. Implicit DDTs are not erroneous, however they enable significant error tolerances. Considering the previous membership testing example of element e {138}. A probe might yield a positive or negative result. If both probes in B as well as $B2$ are positive, the element e {138} is present. Accordingly, in case of two negative results, the overall result is negative. Otherwise, if only B is positive, an element with equal prefix might be present, but not {138}. However, a positive result in $B2$ and a negative result in B indicates erroneous inconsistencies, since an element cannot be existent without its prefix. *Hence, horizontal layers have a error correction characteristic and facilitate a lossy compressed consideration of domains in DTTs.*

DI's balanced binary trees are logically transferred in an implicit structure of DTT with individually – by their prefix – addressable trace trees as nodes, which collapse in functionally comparable traces of size 2^b to cover $b+1$ levels of DIs. Its implicit structure is natively not erroneous, however enables error tolerance by horizontal error correction.

5.4.2. Prefix Hashing and Piecewise-monotone Hash Functions

bloomRF leverages deterministic concepts of trace positions in implicit dyadic trace trees (DTT) (Section 5.4.1) to probabilistically *encode* and compress range information, which is based on the dyadic interval (DI) scheme (introduced in Section 5.3), by setting appropriate bits in an overlapping bit array – similar to BFs (introduced in Section 5.2), as illustrated in Figure 5.5. Generally, standard BFs employ a set of independent hash functions in order to uniformly distribute an element's encoding in the bit array, nevertheless, restructuring of encodings have already been applied for different purposes [CMB+10; DSL+11; LDD11; LNKB19; PSS10]. bloomRF, however, facilitates a broadened predictably valuable applicability in arbitrary membership



Dyadic Interval Scheme (implicit)
0 - 255
128 - 135
128-131
132 - 135
h - intervals
adjacent groups of covering dyadic intervals
128 129 130 131 132 133 134 135
Key
Covering Intervals
Trace-Tree (implicit)
Dyadic Trace Tree (implicit)
Trace-position
Trace
h
Overlapped bit-array
Trace-sized Bit Array Element
Piecewise-monotone hash-functions (PMHF)

Figure 5.5.: bloomRF's encoding of elements is based on concepts of implicit Dyadic Interval (DI) scheme and Dyadic Trace Trees (DTT).

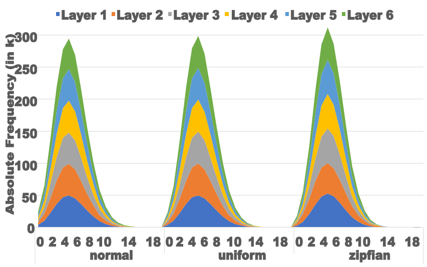
testing of intervals by restructured encoding techniques – i.e. *Prefix Hashing* and *Piecewise Monotone Hash Functions (PMHF)*, as outlined in the following.

Prefix Hashing and Trace Locations. Trace trees represented by a collapsed trace in hierarchical layers of DTTs are individually and directly accessible by a comprised element's prefix. Hence, a set of an element's addressable traces are directly accessible. bloomRF uniformly distributes L hierarchical layers of traces i by independent hash functions f_i on basic trace levels l_i in a bit array of length m. A bit array's length is also expressible by a number of trace locations of words, if m is a multiple of trace width s. As a result, a hash code by f_i for each layer i of an element x can be calculated and referenced to a bit location loc_i in a bloomRF instance.

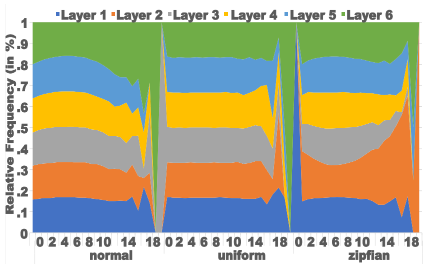
$$loc_i = \left(f_i\,(x >> (l_i + h_i - 1))\,\mathrm{mod}\,\frac{m}{2^{h_i - 1}} \right) << (h_i - 1) \qquad (5.1)$$

Figure 5.6b shows number of trace overlays per word, which exhibit a normal distribution (with an expected peak at 5 to 6 overlays, depicted in Figure 5.6a), and relative share per layer. A logical root is set and suppressed at level 42, yielding 6 layers of height 7 and a trace width of 64. The filter is sized for 10 bits/key and 2 million elements (312500 words) with uniform, normal and unique zipfian[1] distribution. Zipfian distributions have

[1]Zipfian distributions are considered to be accumulations of individual non-duplicated elements.



(a) Absolute Frequency



(b) Relative Frequency [MRBP23]

Figure 5.6.: Overlaid Words per Distribution and Layer in bloomRF.

a negligible increased relative share of overlays at layer 2 (orange), since absolute counts are normally distributed. *Generally, since relative share result in steady proportions over all layers, traces are uniformly distributed over words with sufficient randomization for all prefix hashes.*

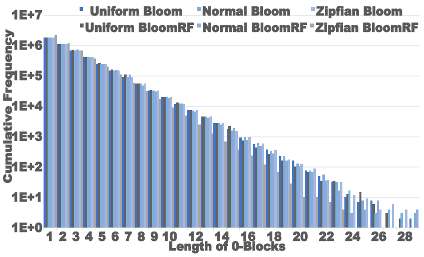
Piecewise Monotone Hash Functions (PMHF) and Element Position. Prefix hashes uniformly distribute traces of DTT's different layers in words of bloomRF's bit array by Equation 5.1. Nevertheless, traces are considered to piecewisely exhibit a sequence of monotonically scattering distribution of comprised elements within occupied words. Hence, respective trace positions of elements must be considered in bloomRF's encodings. By knowing the basic trace level l_i of a layer i, the respective position offset Δ of an element x from loc_i is cheaply gathered by a bitwise AND (&) operation:

$$\Delta = (x >> l_i)\,\&\,\left(2^{h_i} - 1\right) \tag{5.2}$$

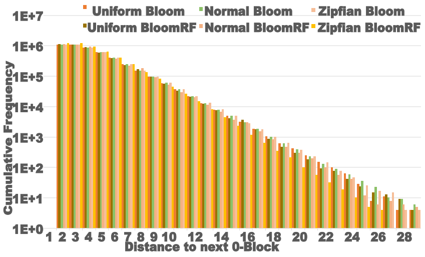
Hence, piecewise monotone hash functions $PMHF_i$ in bloomRF are expressible by a sum of the first bit location of a trace's word in the bit array loc_i and a offset of a specific element Δ:

$$PMHF_i = \left(\left(f_i \left(x >> (l_i + h_i - 1) \right) \bmod \frac{m}{2^{h_i - 1}} \right) << (h_i - 1) \right) + \left((x >> l_i) \,\&\, \left(2^{h_i} - 1 \right) \right) \tag{5.3}$$

Since a family of PMHF is restructuring the random scatter of set element positions, its effects on the bit array also have to be considered. Unset bits are an appropriate metric as they indicate areas, which have never been attained. Significant differences to regular BFs in bloomRF would indicate issues in random scattering properties. Both filters are equally instructed for a number of 2 million elements, i.e. 10 bits/key and for BF optimal 6 hash functions and utilizing 6 PMHF (layers of bloomRF's implicit DTT and a cropped logical root). Different data distributions are tested. As shown in Figure 5.7, the length (Figure 5.7a) as well as the distance (Figure 5.7b) of significant 0-runs of bits behave pretty similar. Zipfian exhibits a slightly differing behavior with an increased number of shorter runs of unset bits but also an increased number of unattained bits, what is explained by the hierarchical hashing of accumulated elements in specific intervals – i.e. lower layers exhibit a continuous sequence of set elements for specific intervals but are empty for others, indicated by overlaid element positions in higher levels due to vertical error correction.



(a) Length of 0-Runs



(b) Distance of 0-Runs

Figure 5.7.: Scatter of Unset Areas in BF and bloomRF. [MRBP23]

5.4.3. Error Correction Policies

Lossy compressed representation in overlaid traces of DDTs in bloomRF's words yield erroneous results, since word positions are not clearly associated to single traces. bloomRF applies different native as well as extensible deterministic and probabilistic error correction policies.

Vertical Error Correction by PMHF. A value of $PMHF_i$ represents a DTT's trace with the base level l_i on a hierarchical layer i. Hence, only one $PMHF_i$ is responsible for an element's representation on level l_i and others operate on prefixes or also on subsequent bits. Nevertheless, as outlined in deterministic arbitrary membership testing of DTTs (Section 5.4.1), several hierarchical layers are testable for deterministic verification, especially in the lossy compressed representation of overlaid traces in bloomRF – i.e. if an element e {138} in Figure 5.4's trace tree $B2$ is present, but not in B, its element position in the word location of $B2$ is not related to $B2$ and erroneous, due to overlaid traces.

In order to measure the effects of vertical error correction (compare Figure 5.8), a bloomRF is instrumented to store elements in traces of size 64 per layer. Words have a fill ratio of 50 to 60% with sporadic outliers of 100%. An inserted element (green) is identified and the entire distance of the lower 3 related layers is tested element by element. Layer 4, 3 and 2 respectively

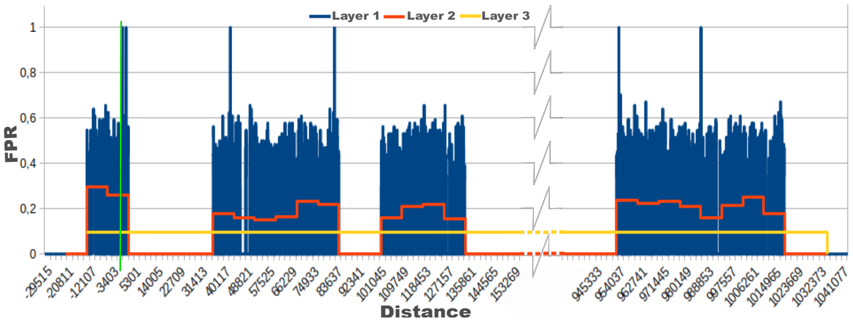


Figure 5.8.: Vertical Error Correction by Hierarchical PMHF.

cover 2^{21}, 2^{14} and 2^7 elements per trace position with a trace tree height of 7. Moreover, each position covers two trace trees in the subordinated layer, whereas one trace in layer 3, 2 and 1 respectively cover a distance of 2^{20}, 2^{13} and 2^6 with its respective word's fill ratio in bloomRF. Hence, the FPR is fluctuating in these frequencies for respective layers.

Whereas layer 1 (blue) generally exhibits an FPR similar to a word's fill factor, it is corrected by layer's 2 (red) unset positions, wherefore its FPR is frequently reduced to 0, yielding an average FPR by layer 2 of 15 to 30%. An equal effect is observed by layer 3 (yellow) with an average of 8% and so forth. *Unset positions of upper layers increasingly correct erroneous areas of overlaid subordinated layers by vertical error correction.*

Flexible Distance of Layers. Effects of deterministic vertical error correction rely on the structure of an underlying implicit DTT. Exponentially growing intervals per layer and reduced number of appropriate $PMHF$ yield diminishing accuracy of deterministic effects. Number of applicable upper layers is increased by smaller trace tree sizes – i.e. less bits of an element prefix are covered per $PMHF$. By this means, a trace tree of $h = 7$ (64 bit trace) can be represented by two layers of $h_a = 3$ (4 bit trace) and $h_b = 4$ (8 bit trace). Thereby, base levels l_i are flexibly applicable for different layers.

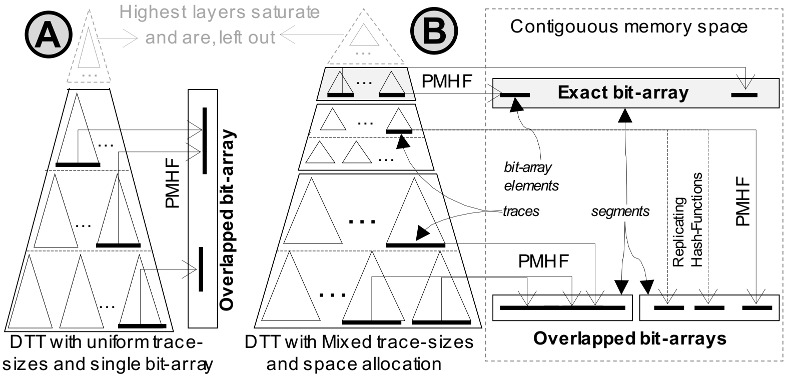

A
Highest layers saturate and are, left out
B
Contigouous memory space
PMHF
Exact bit-array
PMHF
Overlapped bit-array
bit-array elements
traces
segments
Replicating Hash-Functions
PMHF
PMHF
Overlapped bit-arrays
DTT with uniform trace-sizes and single bit-array
DTT with Mixed trace-sizes and space allocation

Figure 5.9.: Vertical Error Correction by Hierarchical PMHF. [RBMP20]

Figure 5.9 shows a DTT with equal distance $PMHF$ Ⓐ and a flexible representative Ⓑ. *Hence, error potential in deterministic probes of elements is flexibly distributable for different layers of an implicit DTT.*

Horizontal Error Correction by Replicating Hash Functions. Errors in upper layers are weighty and virtually impossible to correct by superior layers. Vertical error correction's deterministic effect is improved by probabilistically increased accuracy of upper layers in bloomRF. BFs achieve a probabilistic FPR by repeatedly setting k bits according to the result of independently calculated hashes with the entire element as input. Similarly, bloomRF is able to insert k_i replicated traces for $PMHF_i$ for different layers i. Obviously, a risk of increased fill ratios in words occur, however, every replicating hash function as well as $PMHF$s must satisfy a positive result or the existence of an element is impossible. By this means, there is a significant probabilistic chance to additional exclude not comprised elements from being erroneous positive. Thereby, upper layers are preferably replicated, since range probes tend to utilize less $PMHF$. *Furthermore, excluding large ranges by replicated hash functions likewise improve point as well as range accuracy.*

Memory Segmentation in bloomRF. High number of lower level's set trace positions overlay with upper layers in bloomRF's words and cause weighty errors. Continuous memory space of bit arrays in bloomRF is manageable in order to leverage deterministic characteristics with probabilistic impacts. Upper layers cover large intervals, whereas massive amounts of elements are expressed by single positions. Short prefixes and accumulation of elements yield different hashing behavior. Nevertheless, negative results in these layers prevent large intervals of elements from being erroneous. Hence, error prevention is very valuable in upper layers. As depicted in Figure 5.9 Ⓑ, $PMHF$ and replicating hash function of upper layers are delegated in a separate area of bloomRF's bit array. *Thereby, overlays of traces in words as well as the number of set bit positions per word become manageable as depicted in Figure 5.10, with the result of improved probabilistic characteristics*

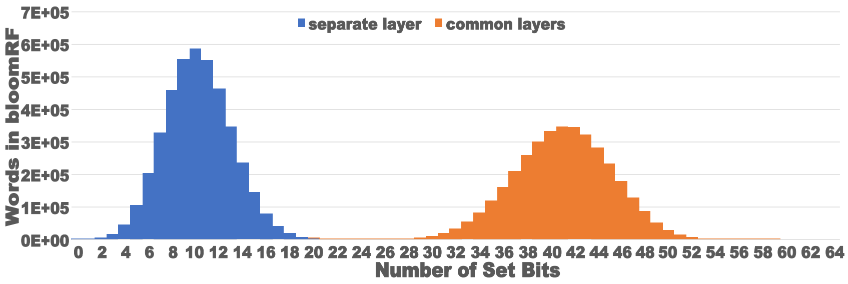


Figure 5.10.: Memory Management in bloomRF improves accuracy of specific Layers.

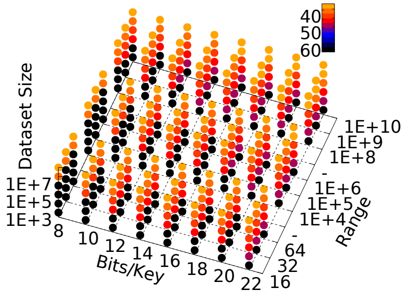
of deterministic approximate membership testing in bloomRF.

Decoupling Deterministic Error Correction from Probabilistic Error Rate. In spite of mentioned error correction techniques, false positive results in upper levels are potentially not tolerable – especially in case of large interval span probes. A drastic approach is to avoid probabilistic errors on a specific valuable layer of the DDT L_{l_e} with the base level l_e as depicted in Figure 5.9. However, upper layers near the logical root tend to saturate, whereas an exact representation is not able to beneficially exclude intervals. On the other hand, lower layers occupy too much space to be entirely stored. Based on few factors, there is a set of probably valuable levels for exact storage. This area is massively increased if considering probably applicable offline optimizations in horizontally partitioned storage and index management structures. For instance, only elements within MV-PBT's partition fence keys require to be represented instead of covering the entire domain. Nevertheless, bloomRF is first of all reflected without application specific optimizations. *Accurate representation of a specific level enables reliable results for large interval spans in bloomRF.*

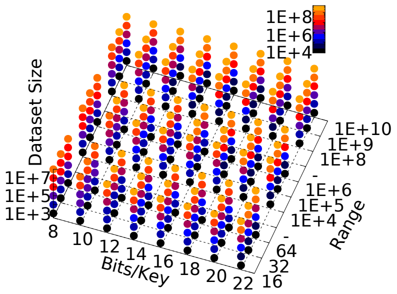
5.4.4. Heuristical Tuning Advisor

Outlined error correction policies in Section 5.4.3 are very powerful, however, their configuration is complex. bloomRF provides a tuning advisor, which is based on heuristics. By this means, bloomRF is as simply applicable as BFs – i.e. by setting basic configuration parameters like the number of elements n, the size in bits/key $\frac{m}{n}$ or absolute size m, but also an approximately estimated maximum probing interval range size R.

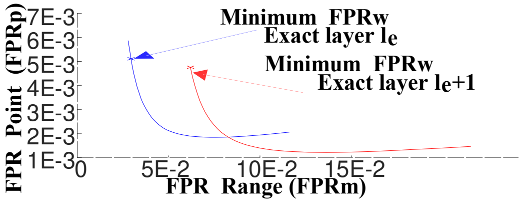
bloomRF is designed to cover a broad area of applicability as real PRF, whereas calculated FPRs of point FPR_p and range FPR_m are weighted (compare Figure 5.11c), hence R indicate a rough area rather than a precise focus like in Rosetta [LCK+20]. As a result, the structure of the DTT is described in a vector of height h_i up to $|H|$ for each layer i and mapped to bloomRF's error correction mechanics of replicated hash functions k_i and



(a) Base Level l_e of Exact Layer L_{l_e}



(b) Size of Memory Segment ms_2



(c) Estimates for fpr_m and fpr_p when varying l_e

Figure 5.11.: bloomRF's Tuning Advisor weights for broad Area of Applicability with default Configuration Parameters.

memory segments $m_i \in \{ms_1, ms_2, ms_3\}$. As depicted in Figure 5.11, base level l_e of exact layer L_{l_e} (Figure 5.11a with a conclusive absolute size ms_1) and memory segment size of ms_2 (Figure 5.11b) vary marginally for a given memory budget m (described by n in the ordinate and bits/key in the abscissa), thus a variety of range query interval sizes are applicable for an equal configuration. Nevertheless, accuracy (fpr_p) of short ranges might suffer from largely configured R and resulting memory segment sizes as depicted in Figure 5.11c.

The exact layer's L_{l_e} base level l_e depends on the memory budget and should be smaller than 60% of the filter size, hence

$$l_e = min\{l | l^{|H|-l} < 0.6 \times m\}, \tag{5.4}$$

whereas different exact level candidates are examined, e.g. l_e, $l_e + 1 \ldots l_{|H|}$. Thereof, h_i, k_i and m_i configuration parameters are gathered. Based on heuristics, lower layer's height is configured to $h_i = 7$, whereas a 64 bit trace size is derived – i.e. the maximum word size. Furthermore, lower layers are typically assigned to ms_3 and do not have replicating hash functions k_i. Intermediate layers between lower and exact ones heuristically require additional accuracy, hence h_i is gradually reduced, k_i is increased and the layer is assigned to ms_2.

The size of ms_2 requires to be determined, since size of ms_1 is deduced from l_e and the size of $ms_3 = m - ms_1 - ms_2$. With the scope on reducing overall FPR up to R, maximum error rate

$$fpr_m = max_{l=0}^{\lfloor log_2(R) \rfloor} fpr_l \tag{5.5}$$

is calculated for comprised DIs and point elements fpr_p. In order to retain PRF characteristic, i.e. not over-optimizing on R, a squared norm with a weighting constant C is build

$$fpr_w = \sqrt{fpr_m^2 + C^2 \times fpr_p^2} \tag{5.6}$$

and the configuration with the lowest fpr_w is selected.

5.4.5. bloomRF's False Positive Rate (FPR) Model

The false positive rate (FPR) in bloomRF must be predictable in order to select beneficial configuration parameters. FPR in BF as well as bloomRF strongly depends on the probability (p) whether an arbitrarily tested bit is still 0 [MU05]. Hence,

$$p \approx e^{-\frac{kn}{m}}$$

$$FPR = (1-p)^k = \left(1 - e^{-\frac{kn}{m}}\right)^k$$

is also applicable for FPR_p in bloomRF, assuming k is the absolute number of $PMHF$ and replicating hash functions for all layers L. By this means, the flexibility of k is limited in a basic configuration of equidistant h_i and might not be optimal as defined in [MU05].

$$k_{opt} = ln(2) \times \frac{m}{n}$$

bloomRF exhibits a similar behavior for fully covering intervals in the dyadic interval scheme (compare J in Figure 5.1). Moreover, partially covering intervals (J^l,J^r in Figure 5.1) are represented as up to two sequences of element positions. In [MRBP23] it is shown, that

$$FPR_m \leq 2 \times \left(1 - e^{-\frac{kn}{m}}\right)^{k - \frac{log_2(R)}{h}} \tag{5.7}$$

Theoretical lower bound FPRs are defined by [CFG+78] for point and [GJLP14] for range. Comparing bloomRF's lower bound with Rosetta [LCK+20] indicates slightly increased memory requirements to achieve similar FPR as Rosetta (compare Figure 5.12a), due to probably suboptimal number of k, however, both come close to the theoretical lower bound. With regards to increasing R in Figure 5.12b, Rosetta is increasingly distancing from the theoretical lower bound, whereas bloomRF converge. Major space benefits are achieved by prefix hashing techniques and incorporation of $PMHF_i$ of up to i layers. Thereby, bloomRF achieves a massive area of applicability

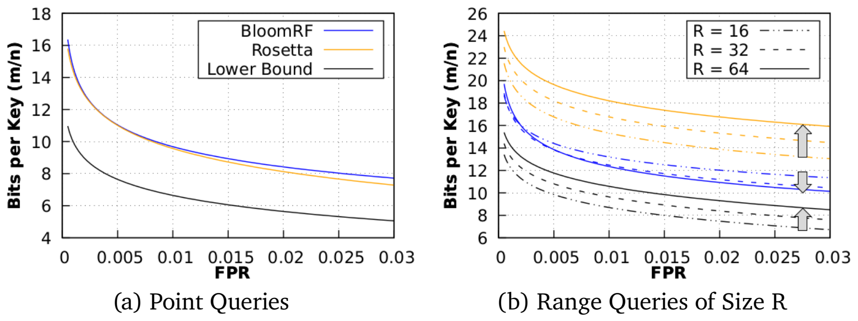


Figure 5.12.: Theoretical lower bound FPR of PRF. [MRBP23]

and is predestined as PRF. Its basic operations are outlined in the following.

5.4.6. Basic Operations

bloomRF is designed to replace point filters and broaden its applicability to arbitrary approximate membership testing in PRF. Facilitating straightforward integration in potential systems, bloomRF satisfies with common configuration parameters n, m or bits/key but enables optimizations on range interval spans up to R on creation of the filter. Moreover, the FPR model enables configuration selection based on target FPR and n. Generally, bloomRF achieves desired behavior by fast low level binary and arithmetic operations.

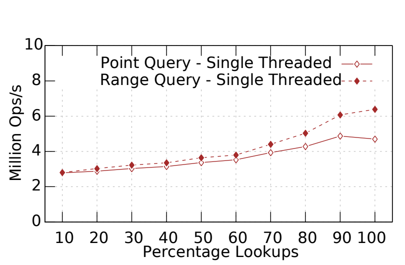
Allocation and Creation. A bit array of size m is zero allocated and a configuration is provided by the tuning advisor. Furthermore, based on the configuration per layer, frequently required information is pre-calculated, e.g. bit masks to parse an element's position in a trace ($(1 << h_{i,r})-1$ in Algorithm 5.1 line 7). In order to represent the DTT and additional information, an optional structure (260 bytes) is allocated in the configurable prototype, though not required in the base configuration. Finally, the filter is ready to get filled with a set of element keys. The dataset might be sorted or unsorted.

Algorithm 5.1 bloomRF: Put

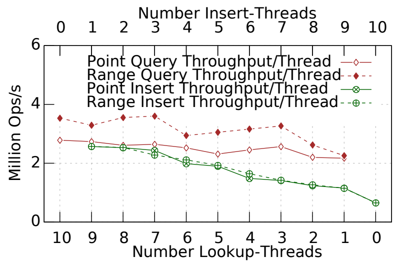
```
function PUT($key$)
    Let $i \leftarrow 0$
    while $(i \leftarrow i+1) \leq L$ do
        Let $r \leftarrow 0$
        do
            Let $loc_{i,r} \leftarrow f_{i,r}\left(key >> \left(l_{i,r} + b_{i,r}\right)\right) \textbf{mod} \left(m >> b_{i,r}\right)$
            Let $mask \leftarrow 1 << \left(\left(key >> l_{i,r}\right) \& \left(1 << h_{i,r}\right) - 1\right)$
            $bit\_array[loc_{i,r}] \leftarrow bit\_array[loc_{i,r}] \quad | \quad mask$
        while $(r \leftarrow r+1) < k_i$
    end while
end function
```

Unsorted Continuous Insertion. An unsorted set of elements can be successively inserted by a `put` operation as outlined in Algorithm 5.1. Generally, calculated bits of a $PMHF_i$ or k_i replicating hash function ($loc_{i,r}$ and $pos_{i,r}$) for an element are successively set like in BFs, however, bloomRF operates on decreasing prefix lengths (bit shift of key by a layer's base level l_{i_r} and the handled bits $b_{i,r}$ in line 6) based on the configuration of the implicit DTT. For each k $PMHF$ and k_i replicating hash functions the location (line 6 indicates the word location contrary to Equation's 5.1 bit position) as well as trace position (line 7) are calculated and the bits are set by a logical `OR` (line 8). This operation might be protected by atomic compare and swap operations, which enable high concurrency and multi-threaded approximate membership testing and online maintenance for a variety of applications.

As depicted in Figure 5.13, by this means, bloomRF exhibits noteworthy multi-threaded throughput per thread. Nevertheless, setting k bits in L layers is more expensive than probing elements. Probes might break on a negative result, as outlined later. Hence, atomic modifications in online operations are very effective. Increasing concurrent modification shrink performance per thread whenever atomic word modifications fail.



(a) Single-Threaded Mixed Workloads



(b) Multi-Threaded Mixed Workloads

Figure 5.13.: Online Characteristics of bloomRF. [MRBP23]

Bulk Creation. PRFs like Rosetta and SuRF rely on comprehensive sorted sets on filter creation. This could probably be a legitimate assumption in RocksDB's [Inc22] LSM-Trees, however, this might be not guaranteed in general and implies additional memory costs of an entirely sorted set on creation as well as narrows its applicability. bloomRF generally relies on single `put` operations, even though bulk creation improvements are feasible. A straightforward memory-sparing optimization is to remember previously inserted elements and skip common prefixes, whereas cache misses and processing are reduced. Furthermore, multi threaded approaches are feasible.

As depicted in Figure 5.14, bloomRF exhibits constant filter creation time for 50 million unsorted integers over different memory budgets. Competitors suffer from additional processing and memory costs to prepare presorted immutable input. Moreover, Rosetta slack off on increasing memory budgets.

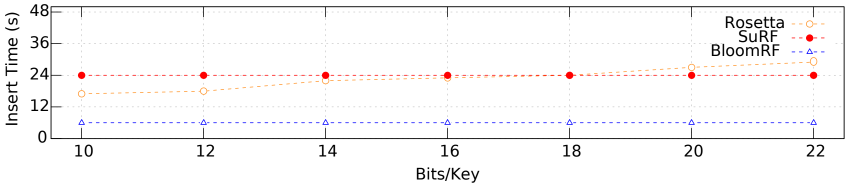


Figure 5.14.: Filter Creation Times for unsorted Datasets. [MRBP23]

Algorithm 5.2 bloomRF: Point Probe

```
1: function PROBE($key$)
2:     Let $i \leftarrow 0$
3:     while $(i \leftarrow i+1) \leq L$ do
4:         Let $r \leftarrow 0$
5:         do
6:             Let $loc_{i,r} \leftarrow f_{i,r}\left(key >> \left(l_{i,r} + b_{i,r}\right)\right) \textbf{mod} \left(m >> b_{i,r}\right)$
7:             Let $mask \leftarrow 1 << \left(\left(key >> l_{i,r}\right) \& \left(1 << h_{i,r}\right) - 1\right)$
8:             if $bit\_array[loc_{i,r}] \& mask = \emptyset$ then
9:                 return $\emptyset$
10:            end if
11:        while $(r \leftarrow r+1) < k_i$
12:    end while
13:    return 1;
14: end function
```

Point Probe of single Elements. bloomRF successively probes single positions within traces in specific bit array locations (as denoted in Algorithm 5.2). Considering error tolerance of the implicit DTT structure, one negative probe (in line 8) disqualifies an element to be related to an entry set. This holds true for $PMHF$s as well as replicating hash functions. Due to individually addressable locations of implicit trace trees by an elements prefix, a deterministic succession of probing points is not necessary. Common prefixes with comprised elements might cause low selectivity and require to probe k probing points. Hence, bloomRF probes points from layer 1 up to L.

Probing Intervals of Arbitrary Range Spans. The reflection of a DTT in bloomRF enables approximate membership testing of arbitrary range interval spans. Contrary to point probes, ranges are successively probed in a deterministic direction from layer L to 1, since selectivity of lowest probed layers might shrink by broadened bit masks and intervals are priorly excluded by upper fully covering layers, hence probing layers $checks$ are initialized with L in Algorithm 5.3 line 3.

Coverings are defined by a common prefix of left l_key and right key

Algorithm 5.3 bloomRF: Range Probe

```
function PROBE($l\_key, r\_key$)
    Let $i \leftarrow L$
    Let $checks \leftarrow$ INIT_CHECKS($l\_key, r\_key, i$)
    while $checks \neq \emptyset$ do
        Let $new\_checks \leftarrow \emptyset$
        for each $check \in checks$ do
            Let $r \leftarrow 0$
            if ISCOVERING($check$) then                ▷ Fully Covering Intervals
                Let $key \leftarrow$ LEFTKEY($check$)
                Proceed similar to Point Probe ...
                if negative result then goto line 6
                finally append determined check(s) in i-1 to $new\_checks$
            else                                       ▷ Partially Covering Intervals
                Let $l\_key \leftarrow$ LEFTKEY($check$)
                Let $r\_key \leftarrow$ RIGHTKEY($check$)
                do
                    $loc_{i,r} \leftarrow f_{i,r}\left(l\_key >> \left(l_{i,r} + b_{i,r}\right)\right) \textbf{mod} \left(m >> b_{i,r}\right)$
                    Let $mask \leftarrow$ GETMASK($l\_lkey, r\_key$)
                    if $bit\_array[loc_{i,r}] \& mask = \emptyset$ then
                        goto line 6
                    end if
                while $(r \leftarrow r + 1) < k_i$
                return 1
            end if
        end for
        $i \leftarrow i - 1$
        $checks \leftarrow new\_checks$
    end while
    return $\emptyset$
end function
```

r_key at the base level of a layer, i.e. only one position in the trace must be set and only one trace is probed per layer. Therefore, l_key and r_key prefixes are equal and it is sufficient to perform a point probe on one prefix. A positive result yield probing in its subjacent layer of a longer common prefix.

Once prefixes differ on base level of a layer i, it belongs to a partially covering interval. This involves up to two traces in a layer, which are appended to *new_checks* – i.e. left and right sibling of a positive position in the upper layer are testable. Before traces are added, a dyadic decomposition is performed on the top level of a layer, since traces are only partially covering the interval. By this means probing intervals are decomposed by its word location in bloomRF.

In a successive iteration, partially covering intervals are probed in their respective location (line 17). Nevertheless, the *mask* for probing positions is a set of continuous bit positions between a respective l_key and r_key. Whenever a partially covering interval cannot be excluded, the filter returns a positive result. Otherwise, further partially covering intervals in the layer are probed until every $check \in checks$ is tested and finally returned.

Major advantage of this approach for MV-PBT is that probes in equal configurations are only calculated once and can be reused in every protected partition. Based on this assumption, two corner cases of this approach are feasible to improve FPR, which are not outlined in Algorithm 5.3. First, an upper parent trace spans two positions but the covered interval affects only inner two traces in the lower layer. By this means, the lower layer traces build a continuous interval and can also be considered in $checks$. A second case behaves similar, if inner traces in between left and right element key are excluded by their parent.

Enabling Successive Process of Doubting. Built upon mentioned principles of excluded partially covering intervals it might be valuable to drill down scattered positive positions in a trace similar to the process of *doubting* in Rosetta [LCK+20]. Probes in deterministic direction are extended towards layer 1 whilst a negative result is probabilistically feasible.

Obviously, successive deterministic exclusion of partially covering intervals cannot rely on singly pre-calculated information, since possible trace locations exponentially grow per layer. Hence, the algorithm starts '*doubting*' whenever a base level of an interval is partially covered by a trace position.

By this means, bloomRF successively puts not yet excluded subordinated intervals to *new_checks* and calculates word locations and bit masks on the fly. Subordinated intervals are determined by the resulting word of a logical `AND` between *mask* and the respective word in the bit array. Doubted and not yet excluded intervals are calculated and added to *new_checks*.

In order to avoid excessive linear growing processing costs like in Rosetta [LCK+20], bloomRF evaluates valuable probes. With regards to Figure 5.10, empty words are unlikely, whereas it is not valuable to probe subordinated layers of consecutive doubting positions. In this case, words are probed against fully set bit masks, which yield a positive result anyways. Moreover, it is feasible to stop doubting in case of excessive work – i.e. if too much subordinated layers are added to *new_checks*. Hence, bloomRF enables a configurable trade-off between cost and performance.

By this means, especially in case of skewed zipfian distributions, bloomRF achieves an improved FPR up to 46% in the library experiments in Section 5.5. Nevertheless, the cheap covering approach is already competitive in general.

5.4.7. Multi Attribute and Data Type Support

Approximate membership testing is probably performed on a variety of datatypes and number of search key attributes in DBMS. Considering the target system MV-PBT of this research, comprised elements are common search key attribute types in storage and index management structures. Therefore, (un-)signed integer and floating point numbers as well as short character strings are considered. Moreover, multi attribute filtering is mandatory in secondary indexing.

bloomRF is based on a fixed sized 64-bit unsigned integer domain for element keys. The principles of bloomRF are generally applicable to any integer domain of iterable elements. Smaller as well as larger domains are feasible, due to the flexible structure of DTT. However, effects on $PMHF$s and number of replicating hash functions must be considered.

On the base of unsigned integer domains, element values of other datatypes are transformed in an equal arrangement of strictly ascending element positions in traces. bloomRF provides access methods for different datatypes. Following challenges emerge.

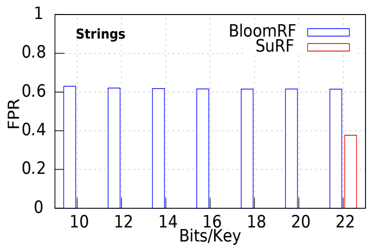
Arrangement of Signed Datatypes. Implicit DTTs in bloomRF are representing positions in traces from a datatypes minimum value (first position) to a maximum value (last position). In signed datatypes, typically a set most significant bit represent negative values, however for conclusive or declarative reasons.

For instance, a 16-bit signed integer maximum value of $2^{15}-1$ ($max = 0x7FFF$) is followed by its negative minimum value $min = 0x8000$ and (-1) ($0xFFFF$) is followed by zero ($0x0000$). Conclusively, every value with a set most significant bit is negative. The domain is shifted to represent values smaller than zero in case of integers.

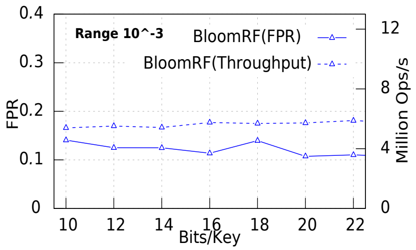
Nevertheless, values of signed datatypes are mapped to its corresponding trace positions by cheap bit operations. In case of integers, an exclusive OR (XOR) on the most significant bit shifts values to its corresponding position – i.e. a 16-bit signed integer's value val is shifted to its respective position pos in a unsigned domain by $pos = val\ XOR\ 0x8000$.

Floating Point Numbers. Floating point numbers are typically signed, too. However, there are certain differences in their representation of decimal numbers to integers. Actually, decimals do not build a sequence of explicit succession, however, floating point numbers are approximately represented by a fixed domain of bits with positive and negative values, declaratively indicated by the most significant bit. Further bits indicate the distance by a mantissa and exponent, whereas negative values are represented in inverted arrangement for positioning in traces. Therefore, bits in negative floating point numbers are inverted and positive are shifted by an exclusive OR similar to integers (compare [MRBP23]).

Other approaches might apply this approach, however, already very short probing intervals result in huge interval spans, which require the massive



(a) Variable Length String



(b) Floating Point Number [MRBP23]

Figure 5.15.: bloomRF efficiently supports Datatypes with constant FPR.

application area of bloomRF. As depicted in Figure 5.15b, bloomRF exhibits constant FPR and throughput in probing intervals of 10^{-3} on NASA's Kepler mission dataset [NAS16].

Fixed Size Strings. Strings subjecting different lexicographical ordering conventions in their fixed sized domain of characters, which might result in undesired behavior for binary treatment – e.g. upper and lower case characters are represented by successions of continuous 8-bit numbers. Avoiding exceptional behavior is incumbent upon the user, since bloomRF operates on bits. However, bitwise ordering in DTT is feasible for a fixed domain size, preferably of a power of two, like 64-bit unsigned integers. In order to map strings in an integer domain, several techniques are feasible, like prefix encoding.

Variable Length Strings. Variable length strings are not limited to a fixed domain, whereas adjacent elements are uncertain. Similar to mathematical decimal numbers, new adjacent elements are identified by appending additional digits – i.e. strings are infinitely extendable by adding characters. In bloomRF, a fixed depth in the DTT is encoded by interval information in traces, on which range probes are possible. Suffixes are hashed and encoded in the bottom layer to enable enough randomization for point probing of the entire string.

As a worst case stress test for bloomRF, filters are created for 1 million abstract portions of DBpedia's Wikipedia Extraction dataset [DBp20] and probed against range intervals of 2^7; results are presented in Figure 5.15a. Rosetta is not designed for this interval range and not tested. SuRF's deterministic approach in the trie representation starts working at a size of 22 bits/key with an FPR of 40%. Even though this is a predestinated area covered by SuRF, its performance is not convincing compared to bloomRF, which achieves 60% FPR by much less bits/key, yielding better cost-performance trade-offs. Offline optimizations might improve deterministic FPR in bloomRF, however, are not yet considered.

Multi Attribute Elements. Secondary indexes probably apply multi attribute search keys. Moreover, cross products of attributes enable further linking applications in DBMS. Flexible numbers of attributes introduce additional ordering and probing complexity, which is covered by bloomRF.

Assuming an element with two attributes $(a \times b)$, possibly the combinations '*point* × *point*', '*point* × *range*', '*range* × *point*' or '*range* × *range*' are probed. The latter is currently not in scope of bloomRF, even though short ranges with low complexity are feasible by manipulating probing bit masks. Other combinations are enabled by inserting concatenated attributes $(a \times b)$ as well as $(b \times a)$ with different $PMHF$s and shifted levels in implicit DTT. By this means, '*point* × *point*' as well as '*point* × *range*' are probed on $(a \times b)$ and '*range* × *point*' is probed on $(b \times a)$. Since the inserted elements are doubled, increased bits/key are necessary.

Actually, '*range* × *point*' is not a basic capability in secondary indexes of type $(a \times b)$. Nevertheless, query optimizers probably decide to use indexes for the range query on a and filter on b, based on selectivity. In horizontally partitioned structures, bloomRF provides data skipping, whereas effort of downstream selection is reduced. Moreover, applicability of bloomRF is not limited to MV-PBT or LSM-Trees.

By all means, probes on multi attribute elements are only reliable in combination, since individual treatment shrinks selectivity and accuracy. For

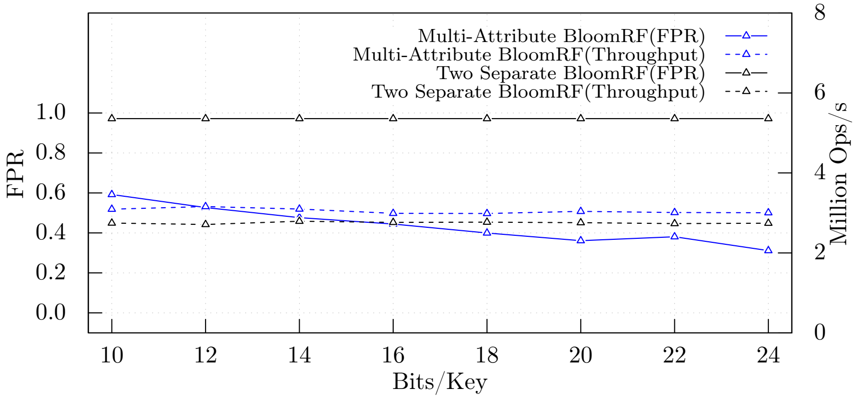


Figure 5.16.: bloomRF's joint effect in Multi Attribute Treatment [MRBP23].

instance, inserting $(a_1 \times b_1)$ and $(a_2 \times b_2)$ would also include $(a_1 \times b_2)$ and $(a_2 \times b_1)$ in individual treatment. This effect is evaluated in Figure 5.16. A multi attribute bloomRF (blue) and two separate bloomRF (black) are built upon $(ObjectID \times Run)$ of Sloan Digital Sky Survey DR16 [DR119]. Inserted values roughly follow a normal distribution. Probes have the pattern '$const \times [0..300]$'. Even with increasing memory budget, separate bloomRF's are not able to achieve reliable results. Moreover, separate treatment is more expensive, with respect to lower throughput. Multi attribute bloomRF achieves much better FPR, whereas selectivity can be improved.

Provided datatype and multi attribute support in bloomRF is an unique technological edge in PRF, whereas its holistic applicability in DBMS and K/V-Stores is feasible.

5.5. Experimental Evaluation of PRF

bloomRF is experimentally evaluated in standalone and system integrated settings. Experiments are performed in the introduced testbed in Section 2.1.3. Several dimensions are considered in latencies and FPR performance

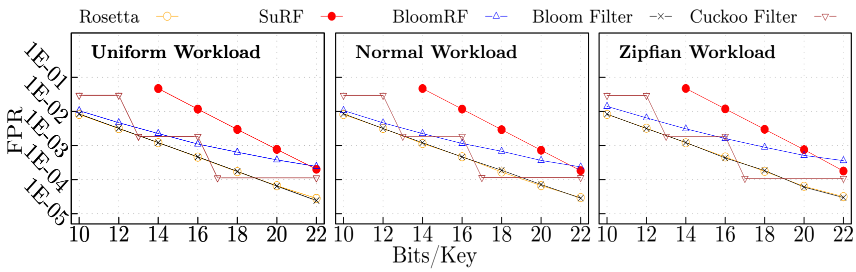


Figure 5.17.: Standalone Point Probe Performance [MRBP23].

of PRF approaches. Datasets and workloads are of non-overlapping uniform distributions, if not defined otherwise. Generally, zipfian distributions are considered to be accumulations of individual non-duplicating elements.

Standalone Point Probe. In Figure 5.17, point probe FPR performance is evaluated for different workload distributions. Filters are created for a dataset size of 2 million uniformly distributed elements and different memory budgets. Cuckoo Filters are introduced as practically better BF alternative. For lack of precise configurability in bits/key, a high occupancy of 95% is selected in order to achieve best possible results. Stored fingerprints are varied in size to keep bits/key equal or smaller than reported parameter settings. SuRF exhibits similar challenges in configuration.

First, bloomRF's competitors except SuRF are only capable or configured for point probes. Cuckoo achieves best FPR at 17 bits/key, however, generally remain below expected performance compared to BFs. Rosetta exhibits similar characteristics to BFs, indicating the use of specialized single level approach [LCK+20] for very short ranges. It is the optimal configuration for the tested point probe case, a mixed point-range workload cause Rosetta to perform worse (configuration for a range interval of 64, bits/key and FPR: 10, 2.66E^{-02}; 12, 1.64E^{-02}; 14, 1.58E^{-02}; 16, 1.16E^{-02}; 18, 8.36E^{-03}; 20, 6.13E^{-03}; 22, 4.42E^{-03}), what actually is not better than SuRF, whereas unified PRF capabilities are limited. SuRF with mixed suffix [ZLL+18] starts working at 14 bits/key and describes the upper bound for point probes.

bloomRF achieves comparable FPR in the efficient area of low bits/key, however, gains less profit from increasing memory budget. Depicted behavior is as expected from the theoretical model in Section 5.4.3. Comparing theoretical lower bound FPR in Figure 5.12, there is a memory budget area, wherein bloomRF is pretty close to Rosetta, but benefit less from additional bits/key. It is a desired behavior of bloomRF as PRF to provide elastic FPR for different interval ranges. As depicted in Figure 5.11, bloomRF's tuning advisor weights configurations pretty similar for a broad area of application, yielding less probabilistic randomization in point probes. Slight performance degeneration in the zipfian workload distribution compared to uniform and normal indicate thereupon. Special tuning configurations of bloomRF might improve point probe performance, since BFs are specialized bloomRFs.

Nevertheless, bloomRF as PRF provides comparable performance characteristics to the widely used competitors in point probes.

Broadened Range Interval Probes. Extensive experimental evaluation in FPR of range interval probes is depicted in Figure 5.18. Due to mass of dimensions and measuring points, a compressed visualization indicates the best PRF by color with absolute distance in FPR to runner-up denoted by symbols. Individual diagrams depict data and workload distribution combinations of *uniform*, *normal* and *zipfian*. Respectively, besides the grouped range interval spans in small ($8-32$) medium (10^4-10^6) and large (10^8-10^{10}) intervals on z-axis, abscissa and ordinate denote the memory budget in bits/key and dataset size in number of elements.

bloomRF (blue) dominates the broaden area of application frequently with massive absolute FPR improvement of more than 10% (denoted as pentagon). Rosetta participate at small interval range spans and occasionally achieves slightly better FPR (yellow, usually less than 0.1%, not more than 1% at best) for uniform and normal data distributions and extremely large memory budgets. Deterministic approach in SuRF is very accurate for range intervals spans covered by exact representation in its trie encoding, whenever the

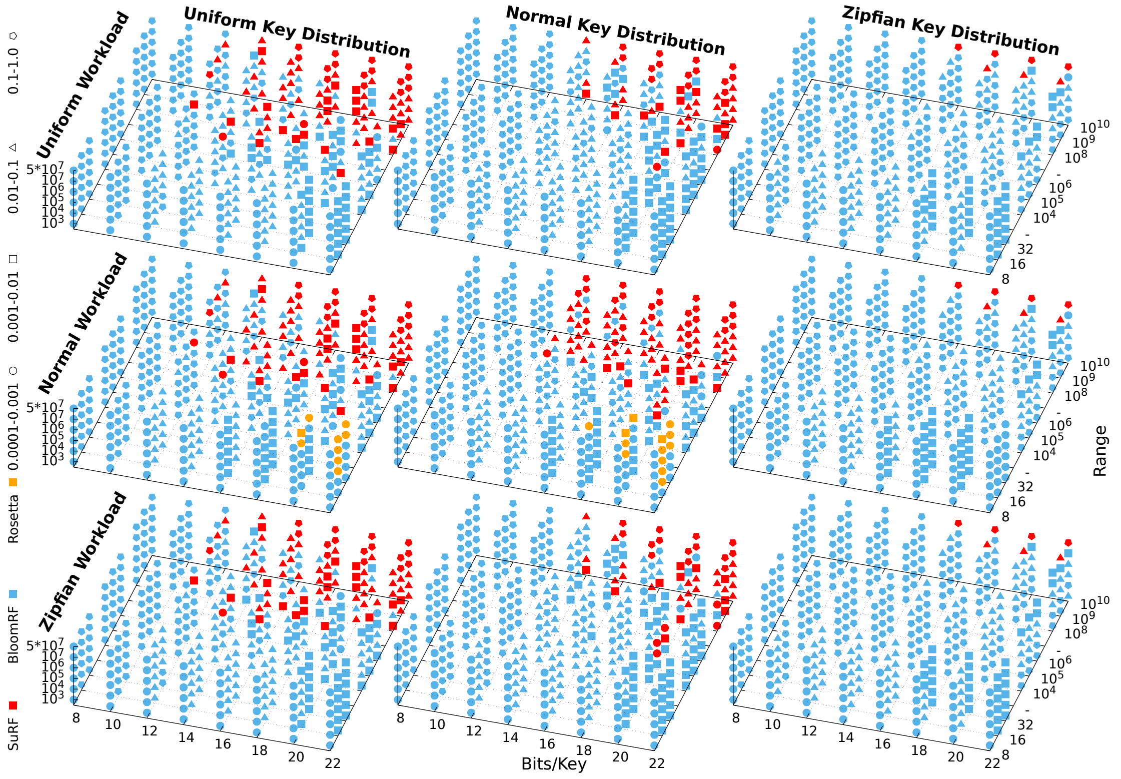


Figure 5.18.: Best Standalone Range Probe FPR for different Data and Workload Distributions.

memory budget is sufficient. For low memory budgets and smaller range interval spans, SuRF does not play any role, since its is not possible to build any trie or meaningful suffixes.

Only bloomRF is capable to efficiently cover the problem space of PRF.

System Integration Benchmarks. bloomRF is implemented in a standalone library, which is integrated besides other PRFs by a filter policy in RocksDB v6.3.6. For persistence it implements its own (de-)serialization mechanism and is placed as regular *full filter block* in each compaction-disabled SST file of a *block-based table format*, whereas LSM-Trees are a sufficient representative for horizontally partitioned storage structures. Range information is provided by *slices* and passed to tested PRFs by an extended interface in the filter policy. Probing points are individually calculated on the fly for each filter instance. Benchmarks are configured to load 50 million uniformly distributed records, yielding 25 components of approximately up to 2.06 million records, if not defined otherwise.

On the left hand of Figure 5.19, absolute latencies and FPR performance metrics are depicted on the ordinates for different memory budgets on the abscissa. Diagrams are categorized in varying range interval spans *small* (8, 16, 32), *medium* (10^4,10^5,10^6) and *large* (10^9,10^{10},10^{11}). Rosetta (yellow) is only present in *small* range intervals, since it is designed for short ranges and already exceeds valuable FPR and latencies for small memory budgets. Moreover, Rosetta is only competitive to bloomRF (blue) for the smallest range on medium and large memory budgets – in FPR as well as latencies. SuRF (red) starts operating for large memory budgets and beats bloomRF only in case of very large range intervals spans. In the horizontally partitioned storage management structure, bloomRF dominates in FPR as well as resulting latencies. On the right hand, FPR of point probes are reported for different workload distributions, similar to Figure 5.17. However, dataset and resulting request distribution depend on the insertion workload in 25 components (protected by fence pointers) and the BF is parametrized by RocksDB, whereas SuRF and BF measurements vary, but

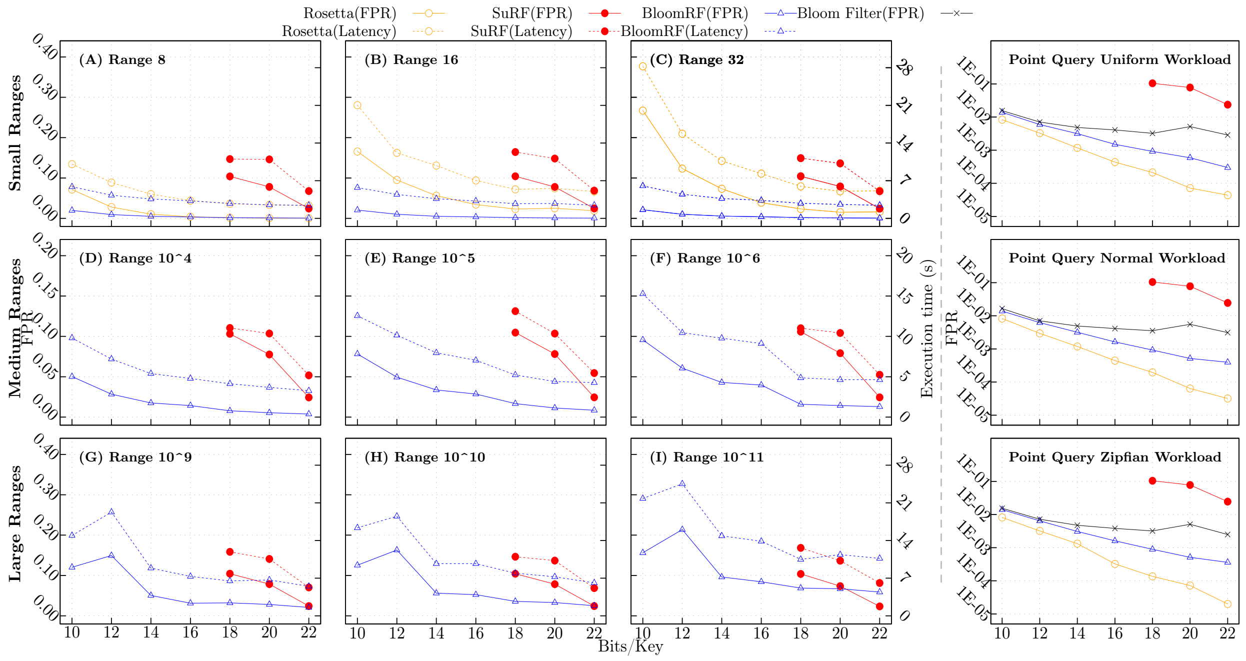


Figure 5.19.: FPR and absolute Latencies of PRF integrated in RocksDB [Inc22] (LSM-Trees). [MRBP23]

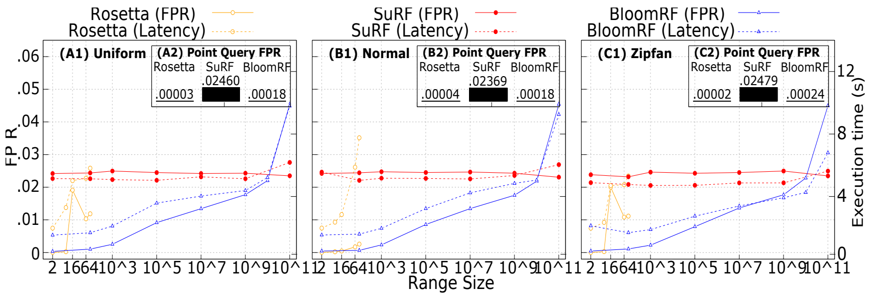


Figure 5.20.: Integrated PRF's FPR and absolute Latencies per Range Interval Span [MRBP23]. According to [LCK+20], Rosetta's results are stated for its limited scope.

are similar to results reported in [LCK+20]. Based on these reported and reconstructed workload properties, bloomRF beats also BFs like Rosetta, though benefit less from increasing memory budgets as expected from the theoretical lower bound FPR.

Spinning-off the configurations of point probes in RocksDB, range interval accuracy and absolute latencies are depicted in Figure 5.20 for large memory budgets of 22 bits/key, hence competitors get sufficient resources to perform well. By these configurations, Rosetta is able to achieve good FPR up to a range of 8 whereupon it generally starts fluctuating in FPR. This is explainable by the inelastic configuration and necessary request distribution in advance of the creation process. Interestingly, in the normal workload distribution, Rosetta remains relatively stable up to range intervals of 64. However, having a look on the latencies, probing costs rapidly increase, indicating excessive effort in the process of *doubting*. Contrary, SuRF remains stable for every range interval, though on a lower performance level.

Only bloomRF covers the entire range interval space with comparable performance metrics in the prime discipline of competitors up to 10^{11}. *Considering Figure 5.21, all PRFs are able to massively reduce request latencies compared to state-of-the-art fence pointers and PBFs – at least in their scope as Rosetta linearly grows by range intervals in request latencies.*

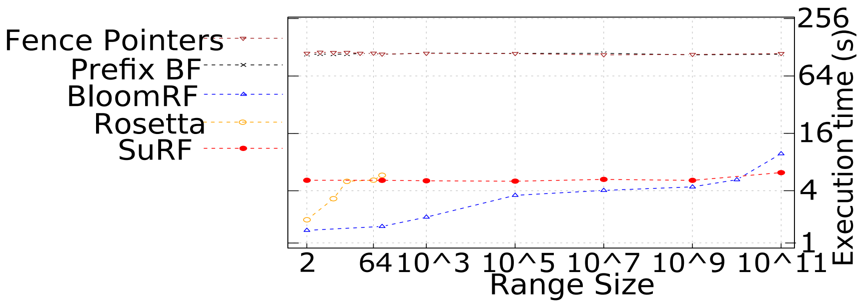


Figure 5.21.: PRFs outperform State-of-the-Art Approaches. [MRBP23]

System Performance Execution Time Breakdown. Negative probes of PRFs in RocksDB prevent protected LSM components from being accessed, whereas unnecessary read I/O and traversal costs are avoided. LSM Trees as a representative of horizontally partitioned storage management structures share this characteristic with MV-PBT (compare Section 4.4.3 Figure 4.28c). Since every performed query on the created 25 components returns an empty result set, a perfect PRF would avoid any read I/O. PRFs are configured for equal memory budgets of 22 bits/key in a normally distributed workload (since competitors perform best, compare Figure 5.20). Moreover, probing points are calculated on the fly for all PRFs, hence cost breakdown and benefits in Figure 5.22 are comparable.

PRFs are arranged across the abscissa with increasing range interval spans. Below, measured FPR is denoted. Stacked bars indicate share on execution time breakdown (ordinate) for different costs. Costs comprise the filter probe (brighter blue) after former deserialization (darker blue) from contents in *full filter block*. Better FPR yield lower read I/O as well as residual processing costs. However, I/O latencies partially overlap with residual processing costs (gray), e.g. by means of pre-fetching, whereas real I/O wait times (red) are reported whenever the CPU idles for I/O. Following observations are made.

First, bloomRF dominates total execution time for every point query and range interval span, except the very short interval of 2. bloomRF's workloads perform 3.4 to 23.7 times faster for point queries and up to 2 times faster

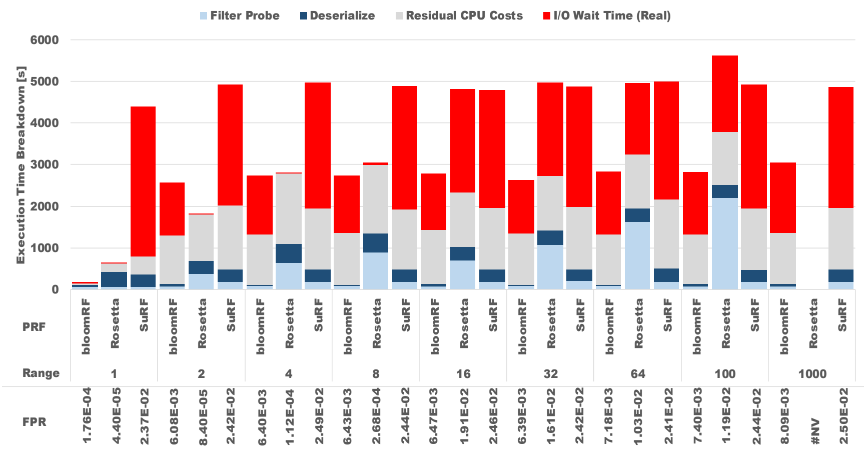


Figure 5.22.: System Integrated PRF's Breakdown. [MRBP23]

for range queries.

Second, bloomRF exhibits predictably constant total execution time for range queries on a comparably best level. SuRF is also constant, though 2 times slower; Rosetta's execution time linearly grows.

Third, CPU times (all except red bars), especially probing costs (brighter blue), exhibit an equal behavior. bloomRF relies on a constant tuning configuration (compare Figure 5.11) – i.e. one configuration handles several range query intervals equally well. Moreover, effort in range probing is also constant, since partially covering intervals only vary by one logical DTT layer[1] and most PMHFs are point probes (compare Algorithm 5.3). SuRF only slightly profit from deterministic accurate trie and mainly operates on suffixes, however in a similar fashion for each range interval. Rosetta apparently operates on a special *single level* configuration for ranges up to 8 – i.e. each element is individually probed. Larger ranges 16 to 100 operate on a *variable level* approach, since probing time slightly decreases from 8 to 16 (and FPR raises), though continuing with linear growth. Obviously,

[1]Assuming a basic bloomRF configuration with a trace width of 64 in lower DTT layers for evaluated range intervals 1 to 1000.

probing costs for 100 are higher than residual processing costs and almost as high as bloomRF's total execution time. 1000 is not reported, since Rosetta is explicitly introduced as PRF for small ranges [LCK+20].

Fourth, although Rosetta exhibits very good FPR characteristics for range intervals up to 8, it is only able to achieve faster total execution times than bloomRF for an interval range of 2. Rosetta's costs per FPR performance are too high to take effect. Moreover, FPR raises for range intervals larger than 8, although it is inelastically configured for fixed range intervals, yielding most expensive probing as well as high I/O costs. SuRF exhibits constant probing costs, however with too low accuracy to take effect. bloomRF's low probing costs and good FPR enable best absolute performance characteristics for a broad range of addressed applicability.

Last, deserialization (darker blue) of contents in *full filter blocks* is required to restore testable filter objects. bloomRF's simple structure of a probably segmented bit array, similar to regular BF, as well as a small DTT-representative meta structure of fixed 260 bytes allow fast deserialization. Complexity in SuRF's as well as Rosetta's structure cause much higher deserialization costs. Admittedly, an additional filter cache might reduce these costs, even though it implies additional complexity and memory costs.

bloomRF exhibits best cost-benefit trade-offs, yielding performance optimal filtering for a broad area of elastic arbitrary applicability.

5.6. Conclusion

Point-range filters (PRF) are valuable auxiliary structures for data skipping in horizontally partitioned storage and index management structures raised by (**RQ3**). Expensive costs of practically inconclusive read I/O, amplified by multiple individually processed traversals, are minimized by cheap and efficient PRF structures.

Cost-benefit trade-offs must not be neglected. Extra costs of PRFs comprise probing by means of processing as well as memory footprints by means of caching. By all means, for positive as well as negative filter probes, these additional costs occur and might adversely affect system performance. Therefore, additional costs of auxiliary filter structures are taken into account by constant probing costs and low memory footprints.

Benefits are specified by the accuracy in false positive rate (FPR) in considerably selective arbitrary query predicates on horizontally partitioned storage and index management structures. In order to avoid redundancies, ideally one related PRF per partition is adequately powerful to cover the entire problem space. Hence, a distinguished cost-benefit ratio is achieved, if arbitrary point and range interval probes on different datatypes and search key attributes are equally considered in design by fixed memory budgets.

Outlined existing PRF approaches exhibit different characteristics with focus on partial aspects in the problem space. Only bloomRF equally facilitates the entire scope of arbitrary approximate membership testing, yielding best cost-benefit ratio. By the nature of horizontally partitioned storage and index management structures, bloomRF exhibits best characteristics to be applied as auxiliary filter structure in MV-PBT.

Chapter 6

Application and System Integration

In this chapter, prototypical system integration details and performance benchmarking evaluation of (MV-)PBT for the application as storage and index management structure are given.

In the first section, OLTP and HTAP system integration benchmarks are performed. It is shown that append-based storage management is beneficial whenever workloads become write-intensive for traditional DBMS designs, however, its performance is limited by traditional B^+-Tree index characteristics – even in case of maintenance cost saving indirection layers. MV-PBT copes with increased indexing effort and enables index-only visibility checks.

Storage management with MV-PBT as a further application (**RQ2**) is evaluated in WiredTiger (WT) [Mon21] – MongoDB's storage engine as well as standalone K/V-Store. WT provides contrastable B^+-Tree and LSM-Tree implementations with modern lock-free maintenance techniques (compare Section 3.1.5.2), whereas a fair and unrestrained comparison of storage management structures is ensured.

6.1. Indexing in Relational Database Management Systems

In this section, several index management aspects of MV-PBT are evaluated in system integration performance benchmarks in a DBMS. Modern workload characteristics are outlined in Section 2.2 and appropriate benchmarks are stated in Section 2.2.1. MV-PBT is designed for a broad range of version-aware applicability, including mixed HTAP workloads with high-rate continuous insertions by OLTP while providing consistent snapshots for OLAP.

In a first instance, a short overview of implementation details and the experimental setup is given. Subsequently, TPC-C benchmarks (DBT-2 [WW21]) are performed for different dataset sizes (scaling factor *warehouses*). Moreover, effects of different approaches are evaluated in a modern HTAP setting (CH-Benchmark [CFG+11]).

6.1.1. Implementation Details of MV-PBT in PostgreSQL with SIAS

MV-PBT is integrated as index management structure in the prototypical append-based DBMS *PostgreSQL 9.0.4 with SIAS* [Got16] (Section 3.1.5.1) – i.e. MV-PBT is based on a very traditional disk-optimized B^+-Tree with 2-byte unsigned integer partition numbers in *partitioned keys* (Section 4.2.1.2), *Cached Meta Structures* (Section 4.2.1.1), physical tuple version reference by *record id*, in-memory garbage collection (Section 4.4.2), partition switch without 8-byte timestamp compression (according to Section 4.2.2) and *bloomRF* (Chapter 5) as auxiliary filter structure.

6.1.2. Experimental Setup

PostgreSQL 9.0.4 (HOT) [Pos21] and the append-based DBMS *PostgreSQL with SIAS* [Got16] are deployed on the *Ubuntu 16.04.7 LTS* server, which is introduced in Section 2.1.3. main-memory is limited to 2 GB, including 600 MB database buffer cache, and the OS page cache is cleaned every second. Base tables and indexes apply a standard page size of $8kB$. They operate on the *Intel DC P3600* enterprise SSD, whereas logging is performed on the

Samsung 860 Pro consumer SSD. Generally, workloads include 30 minutes ramp-up time and 3 hours benchmark runtime, if not stated otherwise.

6.1.3. Evaluation and Selection of Baseline

DBMS spend massive effort to reduce index maintenance costs caused by *rolling entry points* (Section 2.4) in MVCC and online transaction processing (OLTP) by logical indirection with the goal of optimized throughput characteristics in concurrently modifying workloads. However, logical indirection incur accompanying costs, due to unfavorable base table write I/O characteristics (Section 2.3.1), massive memory footprint in limitedly equipped data nodes (Section 2.1) or access delay by indirection.

Traditional Storage and Indexing. For instance, *PostgreSQL 9.0.4 (HOT)* [Pos21] applies *heap-only tuples (HOT)*, which aims for co-location of related tuple version records of a version chain in base tables. B^+-Tree indexes reference co-located *items* on base table pages, which point to old-to-new ordered version record chains. By this means, costly B^+-Tree index maintenance is avoided for *HOT* by sacrificing beneficial base table write I/O characteristics (Section 2.3.1) – especially if workloads become write-intensive.

Traditional Indexing in SIAS. On the other hand, *PostgreSQL with SIAS* [Got16] maintains an indirection layer via VID-mappings (compare Section 3.1.5.1). These potentially incur a massive memory footprint for cached structures, e.g. for 100 warehouses in TPC-C 572 MB[1] are initially necessary to represent mappings for each tuple. If caching is not feasible, read I/O on secondary storage occur to gather missing indirection information.

Non-Robust Performance by HOT. Transactional throughput in the TPC-C-specific metric of new order transactions per minute (NO-tpmC) is depicted for different indirection layer designs (dashed lines) in Figure 6.1 for various

[1]Memory footprint is calculated by the number of initial tuples in the TPC-C scheme [TPC10] (50001100 tuples for 100 warehouses), and 12 bytes per (VID,record id) pair.

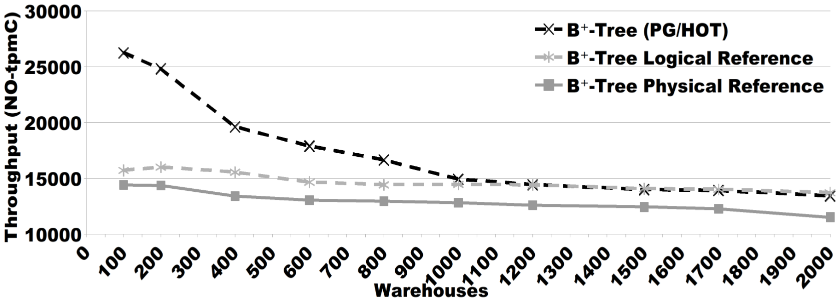
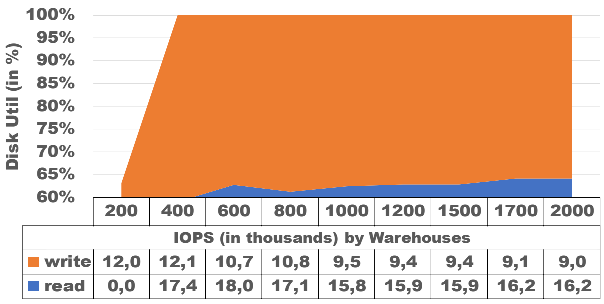


Figure 6.1.: Different Indirection Layer Designs versus Physical Reference [RVGP20]

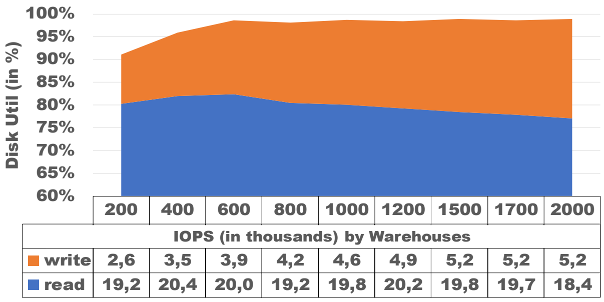
dataset sizes (warehouses). *B^+-Tree (PG/HOT)* (black dashed lines and cross markers) represents *PostgreSQL 9.0.4 (HOT)*. It performs well as long as modified base table pages are mostly located in the sufficiently sized database buffer cache (600 MB) and are rarely evicted to secondary storage. As depicted in Figure 6.2a, *PostgreSQL 9.0.4 (HOT)* exhibits maximum combined read and write I/O per second (IOPS) at a dataset size of 400 warehouses and is entirely I/O bound at 1000 warehouses as IOPS remain stable. Hence, transactional throughput of *B^+-Tree (PG/HOT)* in Figure 6.1 rapidly falls with increasing dataset size, since more randomly modified base table pages are evicted (write I/O) by buffer replacements (read I/O) as a result of the highly skewed workload. *The access pattern exceeds the limits of the Intel DC P3600 enterprise SSD and becomes I/O bound (compare lower mixed random IOPS in Figure 2.2 and operation mix in Figure 6.2a).*

Adverse Write Patterns in Indexes. Nevertheless, the sufficiently sized database buffer cache enable *PostgreSQL 9.0.4 (HOT)* to proceed the skewed workload of transactions on a good level, whereas in [GPHB17] reported performance gains of 30% are not achieved by *PostgreSQL with SIAS* (denoted as B^+-Tree Indirection Layer and depicted with gray dashed lines in Figure 6.1). Nonetheless, SIAS exhibits well-nigh *robust* performance characteristics in every dataset size. Both observations, i.e. robustness on a lower level is



	200	400	600	800	1000	1200	1500	1700	2000
🟧 write	12,0	12,1	10,7	10,8	9,5	9,4	9,4	9,1	9,0
🟦 read	0,0	17,4	18,0	17,1	15,8	15,9	15,9	16,2	16,2

(a) PosgreSQL 9.0.4 (HOT)



	200	400	600	800	1000	1200	1500	1700	2000
🟧 write	2,6	3,5	3,9	4,2	4,6	4,9	5,2	5,2	5,2
🟦 read	19,2	20,4	20,0	19,2	19,8	20,2	19,8	19,7	18,4

(b) PostgreSQL 9.0.4 with SIAS

Figure 6.2.: Disk Utilization in OLTP.

explicable by following facts. First, applied *Enterprise SSD* exhibits specific and increased I/O performance characteristics (compare Figure 2.2) which affect ratios in processing and access latencies (Equation 2.1). Second, memory budgets for database buffer caches are increased and residual RAM is limited, whereas effects of operation system caches and in-memory structures are restricted. Third, visibility checks in SIAS cause successive read accesses and increased RA (compare increased read I/O in Figure 6.2b and 6.2a) due to VID-mapping and spread version records of one logical tuple in base tables – even in typically very short new-to-old ordered version chains in OLTP. Fourth, B^+-Tree indexes still cause random write I/O patterns and shrink overall IOPS performance of an SSD with increased latencies (compare Figure 2.2). Creation of logical tuples and modifications to indexed search key attribute values cause index updates and maintenance as well as yielding random write I/O. Last, average limits of disk I/O utilization are not attained for various dataset sizes (compare Figure 6.2b), indicating excellent characteristics for performance robustness. *For these reasons, throughput is limited by successively performed read latencies from secondary storage devices, which depend on the access patterns.*

In order to avoid indirection layers in OLTP workloads, a naive approach might be to avoid largely sized indirection layers on secondary storage devices. Therefore, physical references of *rolling entry points* (i.e. most recent version records of a logical tuple in new-to-old ordering) are maintained in B^+-Tree indexes (denoted as B^+-Tree Physical Reference and depicted as continuous gray line in Figure 6.1). Predecessor version records remain accessible by successively processing the version chain. By this means, creation of new version records in base tables cause value updates in related index records or an insertion of new index records, if search key attribute values are modified. Amplified modifications in B^+-Tree indexes stress secondary storage devices by increased random write I/O and WA, yielding an average throughput degeneration of 15% (compare Figure 6.1). Hence, traditional B^+-Trees are not capable to maintain physical references to *rolling entry points* in new-to-old ordered version chains.

Since only PostgreSQL with SIAS and logical indirection layers in indexes enables robust performance characteristics and desirable write patterns of base tables on secondary storage devices, it is selected as baseline. Nevertheless, traditional B^+-Tree indexes cause random write I/O, which reduces absolute IOPS performance of SSDs by undesirable access patterns (compare Section 2.1.2) – even though maintenance effort is reduced by indirection layers. These might cause increased memory footprints for VID-mappings and access latencies to entry points of version chains.

6.1.4. Write-optimized Indexing in OLTP

Since the baseline is declared in the previous section, performance effects of write-optimized as well as version-aware indexing in OLTP are evaluated and depicted in Figure 6.3.

PBT outperforms Baseline by 25%. First, a direct comparison of traditional B$^+$-Trees (baseline is denoted as B$^+$-Tree Indirection Layer and depicted in gray dashed lines) and write-optimized indexing with PBT (denoted as PBT Indirection Layer with green dashed lines) is considered. Both apply VID-mappings as indirection layer, hence both approaches are comparably integrated index structures. *PBT Indirection Layer* outperforms *B$^+$-Tree Indirection Layer* by up to 25% for each tested dataset size. Moreover, PBT exhibits even more robustness alongside different dataset sizes. Increased throughput as well as performance robustness is explicable by PBTs write pattern to secondary storage devices with near-optimal WA. Sparsely required index modifications are well-cached in the PBT-Buffer with very low maintenance costs, due to VID-mappings and beneficial sequential writes of saturated partition leaves.

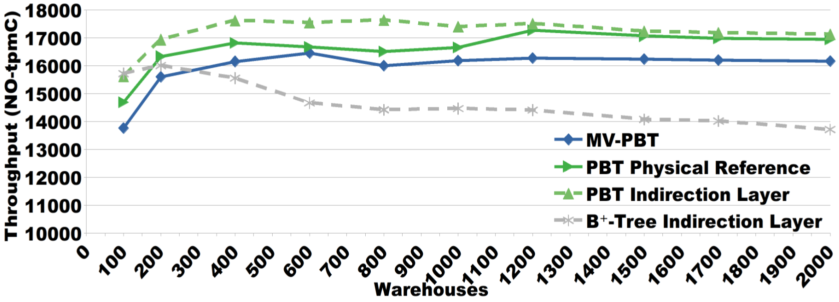
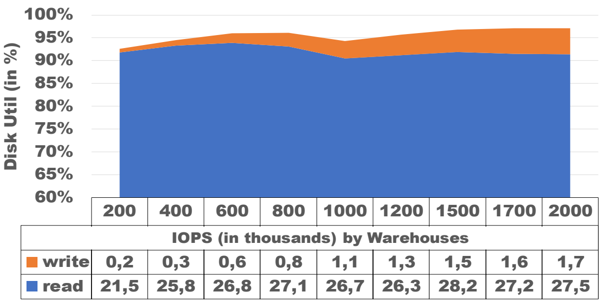


Figure 6.3.: Performance of Indexing Approaches in a TPC-C-like OLTP benchmark [RVGP20]

PBT Scales with Pressure of Rolling Entry Points – 47% **increased Throughput.** In comparison to reduced index maintenance effort by indirection layers, *PBT Physical Reference* (depicted as continuous green line in Figure 6.3) maintains every circumstance of tuple versions by record types (Section 4.2.1.3) according to Section 4.2.3 – i.e. *rolling entry points* are maintained by out-of-place replacements in the PBT. Nevertheless, *PBT Physical Reference* handles maintenance of additional index records very well, since throughput is similar to *PBT Indirection Layer* – especially for dataset sizes larger than 1200 warehouses. Moreover, with respect to *PBT Physical Reference*'s traditional counterpart *B^+-Tree Physical Reference*, a performance gain of 47% is achieved.

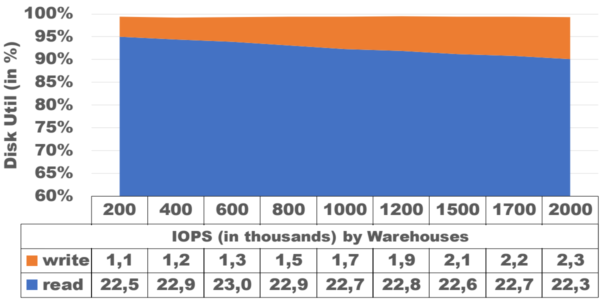
Significant Gains in Robustness and IOPS. *PBT Physical Reference*'s performance drops (compare Figure 6.3) for smaller datasets (100 to 1000) are explainable by concurrency issues in the very hot fraction of the PBT, due to the traditional B^+-Tree techniques in PostgreSQL. This assumption is sustained by *PBT Physical Reference*s disk utilization depicted in Figure 6.4a. Combined near-optimal sequential writes as well as WA in SIAS and PBT yield beneficial access patterns and improve performance characteristics of Flash according to Section 2.1.2. *SIAS with PBT Physical Reference never exceed limits of the Enterprise SSD (maximum disk utilization is 97%) – hence its performance is very robust on a high level even though significant read I/O to secondary storage devices.*

Beneficial Access Patterns optimize Throughput in IOPS. Whilst PostgreSQL with SIAS (*B^+-Tree Indirection Layer*) is able to reduce absolute write IOPS by optimal base table characteristics (Section 2.3), B^+-Trees still cause amplified WA. On comparable transactional throughput (1000 to 2000 warehouses), *B^+-Tree Indirection Layer* reduces write IOPS by only 42% to 52% compared to *B^+-Tree (PG/HOT)* (similar throughput between 1000 and 2000 warehouses, compare Figures 6.2a and 6.2b). *PBT Physical Reference*, however, reduces absolute write IOPS by 81% to 95% on a comparable and higher



	200	400	600	800	1000	1200	1500	1700	2000
IOPS (in thousands) by Warehouses									
🟧 write	0,2	0,3	0,6	0,8	1,1	1,3	1,5	1,6	1,7
🟦 read	21,5	25,8	26,8	27,1	26,7	26,3	28,2	27,2	27,5

(a) SIAS with PBT



	200	400	600	800	1000	1200	1500	1700	2000
IOPS (in thousands) by Warehouses									
🟧 write	1,1	1,2	1,3	1,5	1,7	1,9	2,1	2,2	2,3
🟦 read	22,5	22,9	23,0	22,9	22,7	22,8	22,6	22,7	22,3

(b) SIAS with MV-PBT

Figure 6.4.: Disk Utilization in OLTP with (MV-)PBT indexes.

throughput (compare 600 to 2000 warehouses in Figures 6.2a and 6.4a) and enables strict append-based sequential write patterns for base table as well as indexes. *By this means, Flash SSDs enable increased absolute throughput by beneficial access patterns (compare Section 2.1.2) of approximately* 17.5%.

Version-aware Indexing is Competitive in OLTP. New-to-old version ordering is beneficial in OLTP, since related versions are accessed first (compare Section 2.3.1.2), independent from logical or physical referencing. Successive read accesses on visibility checking are limited (typically less than 3 version records). Hence, index-only visibility checking in MV-PBT aims for competitive performance characteristics to PBT in OLTP (compare Section 4.3.6.2). Whilst successively performed read accesses to version records in base tables are minimized, e.g. non-indexed tuple attribute values could be still required to materialize projections, more reads occur in index structures, due to probably increased index record sizes (compare Section 4.3.6.2). OLTP workload characteristics allow good compressibility of additional timestamps in MV-PBT, though it is not implemented in this prototype (compare Section 6.1.1). MV-PBT applies physical references by nature, whereas performance drops by highly concurrent updates in the very hot fraction of the traditional B^+-Tree implementation are expected for smaller dataset sizes, similar to *PBT Physical Reference*. Actually, MV-PBT achieves almost equal throughput at 600 warehouses. Generally, performance degeneration by index pressure, due to increased record sizes, is at most 5%. According to Section 4.3.1, obsolete transaction timestamps of virtually any record in OLTP is truncatable on partition switch, whereas slightly increased write IOPS (compare Figures 6.4a and 6.4b) are avoided. *Hence, occurring performance drops in indexing physical references and increased write effort by maintenance of timestamps in MV-PBT are preventable by modern B^+-Tree techniques (like in WiredTiger as outlined in Section 3.1.5.2) in the very hot fraction of the most recent partition and timestamp compression on partition switch. By this means, MV-PBT is at least on a par with PBT Logical Reference in OLTP workloads.*

6.1.5. Index-Only Visibility Checks dominate in HTAP

Modern mixed HTAP workloads comprise OLTP with small modifying transactions and long-lasting OLAP queries on a commonly shared dataset instance (compare Section 2.2). In MVCC with SI, logical tuples involve multiple version records in order to calculate a consistent transaction snapshot (Section 2.3). Related version records require to be identified by visibility checks. Therefore, B^+-Tree as well as PBT (both with indirection layer) require to process the version chain in base tables, whereas MV-PBT performs index-only visibility checks (Section 4.3.6). Performance effects in CH-Benchmark [CFG+11] (introduced in Section 2.2.1) are depicted in Figure 6.5. The benchmark is configured with a moderate scale factor (dataset size) of 200, 12 OLTP and 4 OLAP worker threads.

Insight in Competing Factors. Different workload characteristics take effect on system performance. Whilst OLTP transactions might unaffectedly append new version records (compared to basic TPC-C workloads), growing version chains amplify efforts in visibility checks for long-lasting OLAP queries. On the other hand, OLAP incur massive read I/O and increase number of relevant version records, what interplays with available write I/O bandwidth, caching and GC in OLTP. *Different requirements are challenging for index management structures in DBMS that apply MVCC and SI.*

MV-PBT performs best in analytical part (2× throughput). While concurrent OLTP transactions perform updates, small reads and short scans, more complex and long-lasting analytical (OLAP) queries are performed by several worker threads. As time goes by, increasing numbers of successor version records are created by concurrent transaction processing. Version-oblivious index structures identify tuple candidates, i.e. their *entry point references* in base tables, whereupon a downstream visibility check is performed. Reads on base tables incur excessive RA for massive amounts of unrelated version records, yielding poor OLAP characteristics in HTAP (compare B^+-Tree and PBT in Figure 6.5). Analytical scans in B^+-Tree are frequently blocked by concurrently modifying transactions (write locks) and intermingled with more recent tuples, which are not related to analytical transaction's snapshots at all. PBT probably avoids these problems by data skipping of subsequently created partitions, however, visibility checks in base tables incur massive read I/O. MV-PBT is able to skip very hot update-intensive fractions of the index structure by auxiliary filter structures. Moreover, index records of obsolete versions are well removed by different independently performed garbage collection approaches like *Cooperative In-Memory Page Level GC*, due to *Version Chain Discontinuance* (Section 4.4.2), whereas operational costs, SA and RA in MV-PBT are minimized. MV-PBT returns physical references of version records, which are visible to a transaction's snapshot. Its robust search performance is independent from version chain lengths (compare Section 4.3.6.2 Figure 4.21b) and increase throughput by a factor of 2.

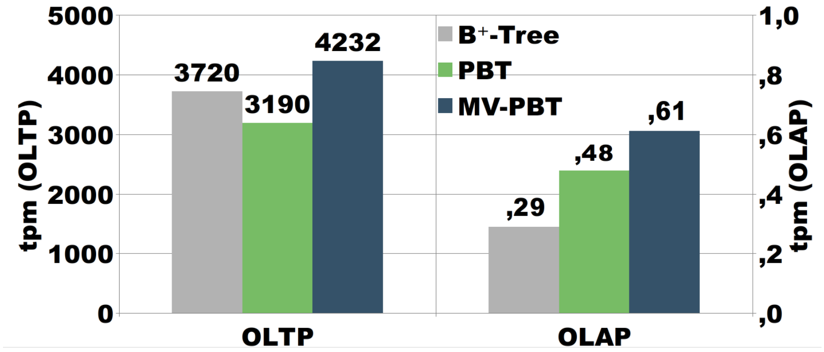


Figure 6.5.: MV-PBT exhibits best Throughput (transactions per minute / tpm) in HTAP Workloads. [RVGP20]

MV-PBT performs best in transaction processing (+14% **throughput).** OLTP transactions are performed similar to the TPC-C workload [TPC10]. One might think, performance characteristics could be similar to TPC-C results depicted in Figure 6.3, however, OLAP queries in HTAP affect workload characteristics by means of impairing cache and GC efficiency. These effects have different impacts on performance of applied index management structures. Pure write-optimization in PBT with the PBT-Buffer loses importance, due to reduced indexing effort by indirection layer, with lowered common buffer and accompanying performance loss compared to B$^+$-Tree. Nevertheless, write-optimization is still relevant in MV-PBT, since every modification is maintained and represented by a version-aware index record for index-only visibility checking. Hence, MV-PBT undertake tasks, which are regularly performed by an intermediate indirection layer. MV-PBT is capable to avoid these indirections and accompanying buffer / cache misses, i.e. access latencies of successively performed read I/O, by inherent index-only visibility checking. Moreover, independently performed GC approaches increase cache efficiency and minimize WA, RA as well as SA, whereas overall performance is increased. Combined effects in MV-PBT improve transaction processing throughput by 14% in HTAP workloads.

6.1.6. Summary

MV-PBT is integrated and evaluated as a version-aware as well as append-based index management structure on top of very traditional and disk-oriented B$^+$-Trees in *PostgreSQL with SIAS*. Both approaches independently facilitate beneficial append-only sequential writes to secondary storage devices of version-aware data. Elimination of random write I/O increase available bandwidth (IOPS) of Flash, due to beneficial access patterns and improved latencies to requested information, which is comprised in massive amounts of persistent data on secondary storage.

Index-only visibility checks in MV-PBT avoid successive indirections to related version records in base tables and enable robust query performance – independent from version chain lengths. Principally, index management with MV-PBT improves system performance in HTAP as well as simple OLTP workloads. However, applied techniques in very traditional disk-oriented B$^+$-Trees in *PostgreSQL 9.0.4* – which MV-PBT is built upon – limit in-memory performance in the very hot fraction of the most recent partition. Nevertheless, improvements and robustness in transactional throughput by version-aware and hardware-leveraging index management is evident for aspired workload characteristics.

In order to evaluate its potentials with modern B$^+$-Tree-techniques, MV-PBT is integrated in *WiredTiger 10.0.1 (WT)* [Mon21] and evaluated with cloud-serving YCSB benchmark [CST+10; RKD21] workloads in the next section.

6.2. Storage Manager in Key/Value-Stores

In this section, MV-PBT's storage management characteristics are evaluated and compared to memory-optimized B$^+$-Trees and the widely used LSM-Trees (with leveled data layout) in *WiredTiger 10.0.1 (WT)* [Mon21] (introduced in Section 3.1.5.2). The basic cloud-serving YCSB benchmark workloads [CST+10; RKD21] (introduced in Section 2.2.1) are appropriated to evaluate different performance aspects of MV-PBT in modern workloads with massive amounts of data on secondary storage devices (compare Section 2.2).

Research objectives are defined as follows. First, MV-PBT is evaluated on top of a memory-optimized B^+-Tree structure in order to derive potentials in database index management with modern B^+-Tree techniques. Major performance gains are assumed by lock-free in-memory optimizations, due to the very hot and update-intensive most recent partition of a MV-PBT. Second, assumptions about related structures made in Chapter 3 are evaluated in a consistent code base with equal techniques. Hence, an ideal environment is given to evaluate bare characteristics of B^+-Trees, LSM-Trees and MV-PBT. Last, storage management is a further possible application of MV-PBT raised by **RQ2**.

6.2.1. Implementation Details of MV-PBT in WT

WiredTiger (WT) [Mon21] is the default storage engine in MongoDB's document store and is applicable as standalone K/V-Store. WT is built upon a lock-free and in-memory optimized B^+-Tree implementation for storage and index management as outlined in Section 3.1.5.2. Built upon B^+-Trees, leveled LSM-Trees are applicable as equivalent storage management structure in write-intensive workloads.

MV-PBT is integrated as storage management structure in *WiredTiger 10.0.1 (WT)* [Mon21] upon the existing in-memory optimized B^+-Tree-structure and applied *Cached Meta Structures* (Section 4.2.1.1). *Partitioned keys* apply 2-byte unsigned integers in *by reference strings* as *partition numbers* (Section 4.2.1.2), which are usually compressed by provided *prefix compression* techniques. *Transaction timestamps* are maintained in in-memory representations of different record types (compare Section 4.3.1) and are discarded on eviction. They are probably appended as additional column if necessary, however, WT generally prevents eviction of active transaction's read- and write-set-related data. The *MV-PBT-Buffer* (Section 4.2.1.4) allows flexible growth of *hazard pointer* referenced in-memory leaves of the most recent mutable partition up to 20% of the buffer share. Partitions are switched and flushed by a tree-walk-based reconciliation process (Section

4.2.2). *bloomRF* (Chapter 5) is included as PRF for approximate membership testing. Moreover, *cached partitions* (Section 4.4.1) with partition number reference are applied every 20 partitions and *consolidated GC partitions* (Section 4.4.2) are generated every 400 partitions by background worker processes.

6.2.2. Implications on Secondary Indexing MV-PBT Stores

WiredTiger (WT) extends basic functionalities of traditional K/V-Stores by several capabilities known from DBMS. For instance, WT natively implements transaction management, schema support, secondary indexing or join operations, whereby it is an appropriate storage engine in MongoDB [Mon21]. Generally, in K/V-Stores key-searchable storage management structures are very popular, e.g. by hashing or key-sorted LSM-Trees and B^+-Trees. By this means, search operations, which are favorably performed on primary keys, and storage management are handled at once – without maintenance and searches in auxiliary primary index structures, like in heap-based storage management (e.g. SIAS or HOT in PostgreSQL).

MV-PBT as storage management structure contains every version record of logical tuples, which are related to any active snapshot, nevertheless, they are probably not located at their final location. B^+-Tree maintenance operations possibly move records across nodes to sustain sort orders – hence, this characteristic is also valid for the most recent partition in MV-PBT. For this reason, unlike to SIAS base table organization, physical referencing by record id (page and slot numbers) in secondary indexing is challenging. On the other hand, MV-PBT provides good search performance by known search key attribute values (Equation 4.3 in Section 4.2.4). Version-aware secondary indexes (e.g. MV-PBT) probably include the entire immutable primary search key attribute values (including partition number) as logical reference. Thereby is the number of searched partitions ($f_{(P)}$) equal to 1 without filter probes. Primary search key lengths must be considered to be as small as possible, since they are included in secondary indexes.

MV-PBT is capable to serve as sole storage and index management structure in WT – especially if most querying operations are performed on primary keys. MV-PBT provides good inherent search and storage capabilities. Searches on secondary indexes cause one additional root-to-leaf traversal in the primary storage structure, whereas version records in SIAS are directly accessible by record id. However, searches on primary key are preferably supported by an additionally maintained primary index, due to its cost model (Section 2.3.2). *MV-PBT provides challenging storage management capabilities to SIAS by inherent search functionalities, however, secondary indexing requires special care in costs of logical references.*

6.2.3. Experimental Setup

WiredTiger 10.0.1 (WT) [Mon21] and the prototypical implementation of MV-PBT in WT are deployed on the *Ubuntu 16.04.7 LTS* server, which is introduced in Section 2.1.3. main-memory is limited to 2 GB, with 200 MB buffer cache (including 20% MV-PBT-Buffer or LSM chunks, respectively). The YCSB framework [CST+10; RKD21] is appropriated for experimental evaluation. Initially, the WT K/V-Stores are loaded with approximately 50 GB with 1 kB value size, unless stated otherwise. Workloads are executed for 3 to 10 hours by up to 6 worker threads in order to keep enough resources for background workers in WT. Prefix truncation and snappy compression are enabled. Direct I/O is enabled and the OS page cache is cleaned every second in order to ensure repeatable, reliable and even conservative results. Experiments are performed on the *Intel DC P3600* enterprise SSD as well as on the *Samsung 860 Pro* consumer SSD (compare Section 2.1.3).

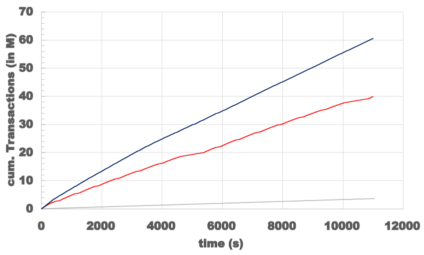
6.2.4. Experimental Evaluation

In this section, basic storage management structures in WT – i.e. in-memory designed B^+-Trees and write-optimized leveled LSM-Trees – as well as MV-PBT are evaluated in standard YCSB workloads. All structures apply equal B^+-Tree techniques and share a common code base, whereas an ideal opportunity to compare structural effects is enabled.

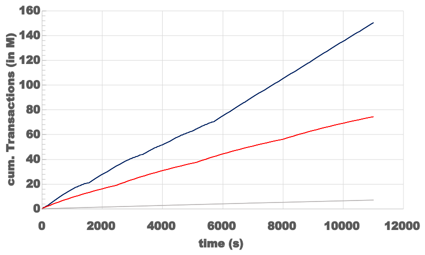
First, modifying standard YCSB workloads are performed for 3 hours and configured as outlined in Section 6.2.3. Results are depicted in Figure 6.6. By default, B^+-Trees, LSM-Trees and MV-PBT are respectively denoted by gray, red and blue lines. Figures depict the cumulatively executed transactions in million per elapsed execution time in seconds. Following observations are made:

MV-PBT exhibits best or at least competitive performance characteristics. Update-intensive workloads (e.g. YCSB Workload A with 50% reads and updates respectively) are the comfort zone of the widely used LSM-Tree. MV-PBT outperforms LSM-Trees by factor 1.5 (Figure 6.6a) to 2 (Figure 6.6b) in the write-intensive setting. B^+-Trees are outperformed by orders of magnitude (up to 43×), due to maintenance operations, massive WA and adversely performed random write patterns. MV-PBT is able to absorb updates in its most recent partition and performs beneficial sequential writes with a minimal WA. *Cached Partitions* (Section 4.4.1) as well as commonly utilized and buffered inner nodes (compare costs statements in Section 4.2.4) enable robust search performance and minimize merge effort and accompanying WA compared to LSM-Trees. These effects are also valid for less update-intensive workloads (YCSB Workload B with 95% reads and 5% updates), whereby MV-PBT outperforms LSM-Trees by a factor of up to 1.7× and B^+-Trees up to 3×.

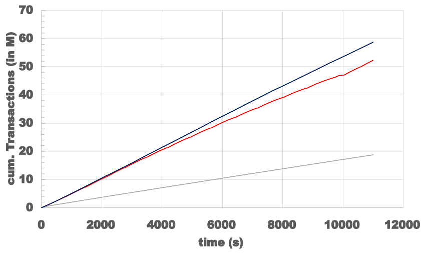
YCSB Workload D (Figures 6.6e and 6.6f) differs from Workload B by the request distribution (95% reads are mainly performed on *recently inserted* 5% records, comprising many empty result sets). Storage management structures are affected by these variations in different ways. Since read operations are preferably performed on recently inserted records, their referencing as well as comprising nodes in all tree structures are very likely to be cached. Nevertheless, several read operations cannot be immediately answered by a valid tuple value, since recently inserted records are not related to a query's transaction snapshot or insertions are still pending by the concurrent workload. B^+-Trees perform best, since they identify nonexistence



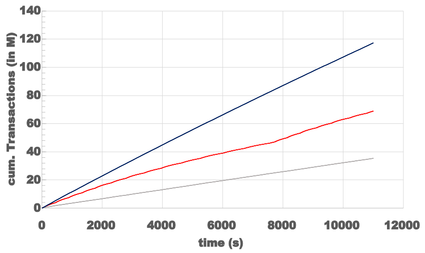
(a) Workload A Consumer SSD



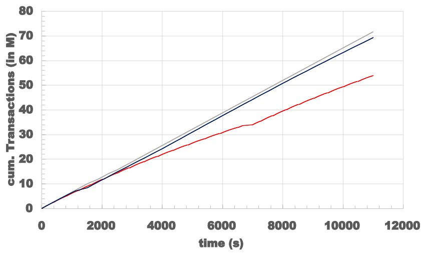
(b) Workload A Enterprise SSD



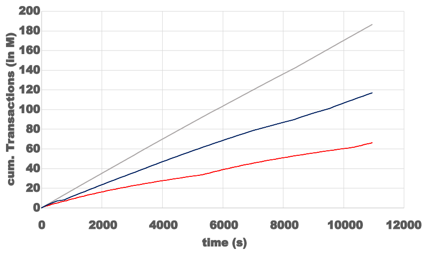
(c) Workload B Consumer SSD



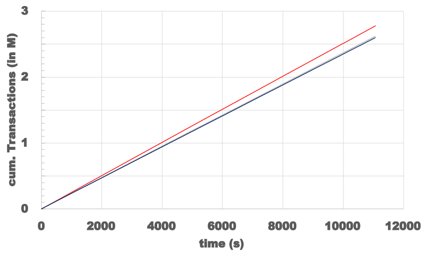
(d) Workload B Enterprise SSD



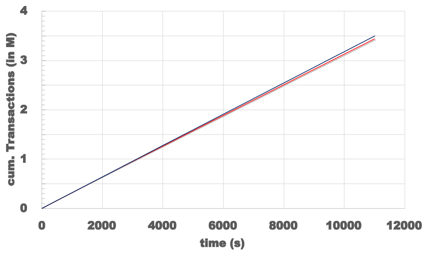
(e) Workload D Consumer SSD



(f) Workload D Enterprise SSD



(g) Workload E Consumer SSD



(h) Workload E Enterprise SSD

Figure 6.6.: Cumulated Transactions per Time in modifying YCSB Workloads

as well as definitively co-located predecessor version records within *one* root to leaf traversal operation with excellent cache probability. However, in case of horizontally partitioned storage structures, version records are probably located at several storage locations. Search operations on nonexistent or predecessor version records *require other components / partitions* as well as their auxiliary filters to be probed. MV-PBT deals much better with partition traversals than LSM-Trees (due to *Cached Partitions* and commonly cached inner nodes), whereas they remain more competitive to B$^+$-Trees and might catch up with more aggressive defragmentation.

In scan operations (Workload E in Figures 6.6g and 6.6h), successively performed read I/O on leaves dominate throughput characteristics. Additional traversals by horizontally partitioned storage management structures have low effects on overall scan performance and all structures exhibit comparable throughput – however, this characteristic depends on the record value size (compare Figure 6.9e). *Considering modern workload characteristics (Section 2.2), MV-PBT exhibits best structural properties for a broad range of applicability and outperforms its competitors.*

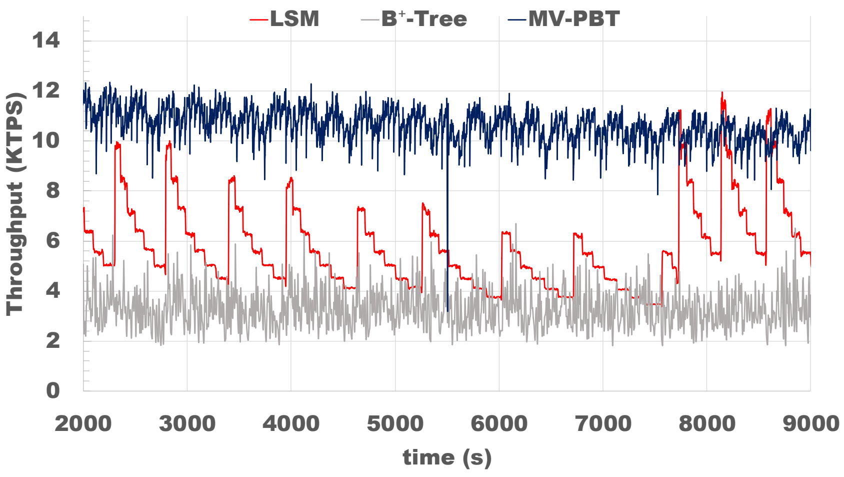
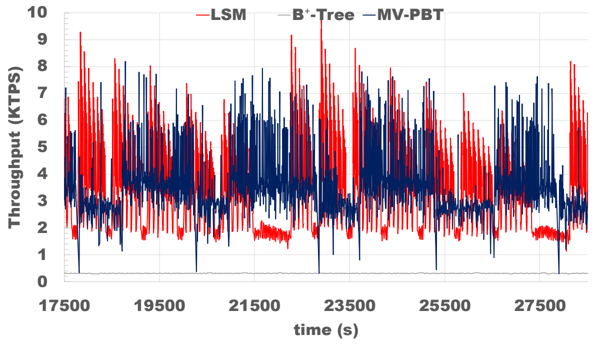


Figure 6.7.: Performance Robustness in MV-PBT.

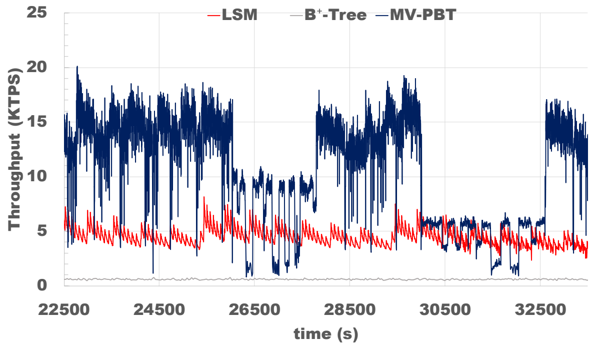
MV-PBT exhibits robust steady performance characteristics. Modifying operations in workloads, like YCSB Workloads in Figure 6.6, affect the conditions of storage management structures and underlying secondary storage devices. In B$^+$-Trees, for instance, an insertion might cause node splits and increase SA. On the other hand, LSM-Trees and MV-PBT generate additional horizontal partitions. Moreover, modifications on secondary storage devices entail inherent operations, which might affect available IOPS (compare Section 2.1.2).

Generally, MV-PBT exhibits very robust performance characteristics (compare steadily increasing cumulated transactions in Figure 6.6). Strictly performed beneficial sequential write patterns (Section 4.2.2), *Cached Partitions* (Section 4.4.1) as well as beneficial GC techniques (Section 4.4.2) allow performant conditions in the storage management structure and leverage secondary storage devices. Exemplary, average throughput per second of applied storage management structures in YCSB Workload B are depicted in Figure 6.7. Whilst the leveled LSM-Tree gradually varies in actual throughput based on its merge status of leveled components, B$^+$-Trees and MV-PBT exhibit very robust throughput – however, MV-PBT performs at least 3 times better. Commonly cached inner nodes enable fast traversal operations and *Cached Partitions* efficiently locate related partitions with the result of robust throughput characteristics. Nevertheless, performance slightly degenerates until covering CP are created or evolving fragmentation is reduced by GC processes. *Even in steady workload conditions after hours of execution, MV-PBT's performance characteristics are very robust (compare YCSB Workload A in Figure 6.8), indicating adequate structural properties, GC operations and beneficial access patterns – leveraging secondary storage devices.*

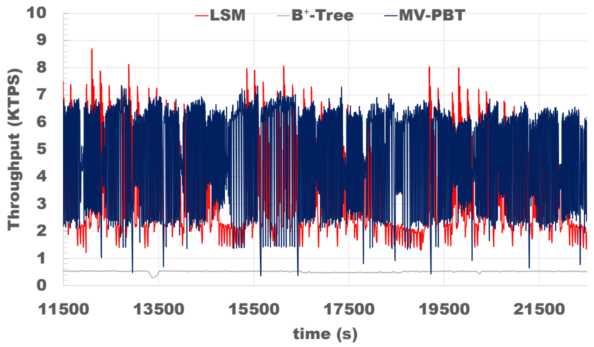
MV-PBT leverages available hardware resources. Datasets are assumed to be update-intensive and large, exceeding the capacity of main-memory. Whilst B$^+$-Trees exhibit poor performance characteristics, if its working sets massively exceed RAM, MV-PBT leverages available hardware resources and turn additional bandwidth and buffer caches in performance. LSM-Trees are



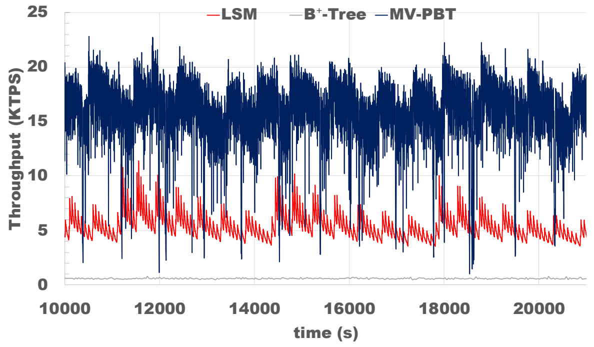
(a) 100 MB Consumer SSD



(b) 100 MB Enterprise SSD



(c) 200 MB Consumer SSD



(d) 200 MB Enterprise SSD

Figure 6.8.: Steady Throughput for Consumer and Enterprise SSD for different Buffer Sizes in YCSB Workload A (after 60 / 200 million transactions in MV-PBT respectively).

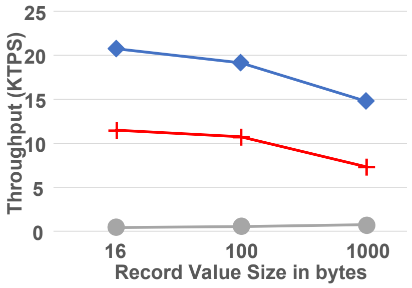
not able to gain performance, due to structural conditions.

YCSB Workload A is performed for several hours to attain steady performance characteristics and is depicted in Figure 6.8. Available hardware resources are varied between 100 (Figures 6.8a and 6.8b) and 200 MB buffer cache (Figures 6.8c and 6.8d) and from Consumer (Figures 6.8a and 6.8c) to much more performant Enterprise SSD (Figures 6.8b and 6.8d). Steady throughput characteristics are assumed, if 60 million transactions in MV-PBT on Consumer SSD and 200 million transactions in MV-PBT on Enterprise SSD are performed.

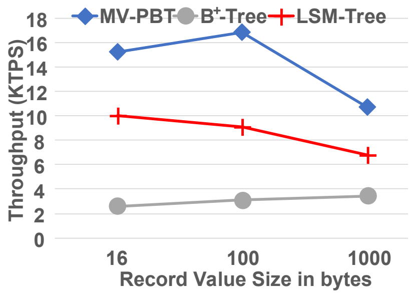
Whilst MV-PBT turn doubled buffer cache as well as additional I/O bandwidth in increased and more robust average throughput per second, LSM-Trees as well as B$^+$-Trees almost take no benefit from additional resources. *These observations indicate beneficial structural and hardware-leveraging properties in MV-PBT.*

Application of Different Record Sizes. Record sizes affect the performance characteristics of storage and index management structures. With increasing record sizes, less records take place in one leaf node. Whilst index record values contain small physical or logical references (e.g. record id), storage management structures probably vary in comprised data lengths from small to large record value sizes. Therefore, YCSB allows to vary the comprised data value lengths. YCSB basic workloads (Figure 6.9) are performed on small (16 bytes), medium (100 bytes) and large (1000 bytes) record value lengths, the initial load is adjusted to match approximately 50 GB dataset sizes.

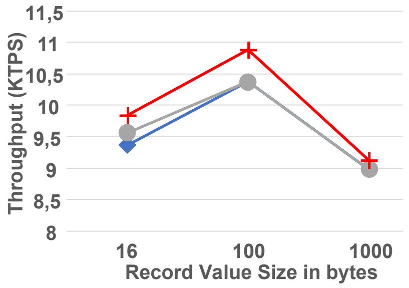
Extensive experimental evaluations are performed on introduced storage management structures. MV-PBT exhibits robust performance characteristics and a broad range of applicability, whereas it generally outperforms its competitors in their respective scope.



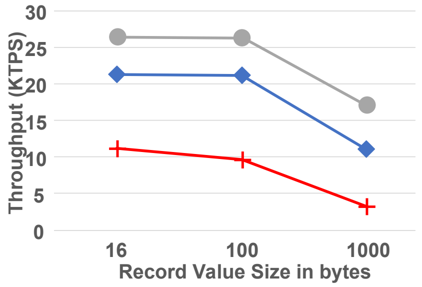
(a) YCSB Workload A



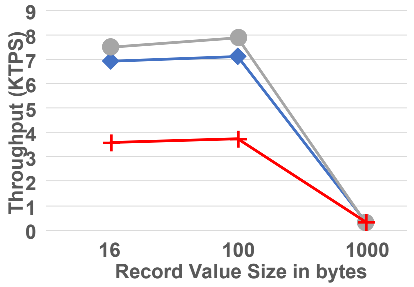
(b) YCSB Workload B



(c) YCSB Workload C



(d) YCSB Workload D



(e) YCSB Workload E

Figure 6.9.: YCSB performance evaluation for different value sizes. [RP22]

Experiments (depicted in Figure 6.9) show similar performance characteristics for every dataset size – especially in updating workloads A (Figure 6.9a) and B (Figure 6.9b) MV-PBT dominates its competitors. This means that determined advantages of MV-PBT (blue) for large record value sizes are also valid for small record value sizes, i.e. likewise in update-intensive multi-version index management with physical reference record ids values MV-PBT outperforms LSM-Trees (red) by up to 2× and the baseline index management structure B^+-Trees (gray) by orders of magnitude.

For the sake of completeness, YCSB Workload C (depicted in Figure 6.9c) comprise of 100% read operations on the initially loaded dataset of 50 GB, hence every storage management structure is in a read-optimized layout. Thereby, MV-PBT, LSM-Trees and B^+-Trees exhibit comparable performance characteristics. This observation confirms beneficial properties of dense-packed and read-optimized layout (compare Sections 4.2.2) with diminishing costs of additional partition numbers (compare Section 4.2.1.2). However, as known from Section 4.4.1 Figure 4.23 as well as Figure 6.7, LSM-Trees are not capable to keeping up throughput in case of fragmented components. Even B^+-Tree's performance characteristics slightly degenerate by modifications and MV-PBT sustain competitive and robust read performance. As a result, MV-PBT's performance drops in the search-intensive YCSB Workload D (Figure 6.9d) are much less incisive than in LSM-Trees.

Finally, MV-PBT achieves comparable performance to B^+-Trees in YCSB Workload E (5% insert and 95% scan) depicted in Figure 6.9e. Whereas all storage management structures perform equal for large value sizes, LSM-Trees fall behind for small and medium value sizes. Successively performed read I/O on data access in leaves stop dominating costs and traversal operations become more important. *Cached Partitions* and commonly cached inner nodes enable cheap merge sort scan operations in MV-PBT, whereas they remain competitive to the ubiquitous B^+-Trees.

6.2.5. Summary

MV-PBT is integrated as append-based and version-aware storage management structure in the K/V-Store *WiredTiger 10.0.1 (WT)* [Mon21]. WT implements LSM-Trees and B$^+$-Trees as well on a shared code base with commonly applied modern B$^+$-Tree-techniques – an ideal opportunity for structural comparison is given. B$^+$-Trees hardly take profit from in-memory optimizations in WT, since every update immediately result in write I/O to secondary storage devices (compare Section 3.1.1 Equation 3.6). However, B$^+$-Trees are able to profit from cheap logarithmic complexity in search and scan operations (compare Section 3.1.1 Equation 3.4), yielding good performance properties in YCSB Workload C, D and E. LSM-Trees enable beneficial sequential writes to secondary storage devices with a low WA, however, search operations in the fragmented components operate on separately managed inner nodes, yielding poor search operation performance (compare Section 3.1.3). Frequent merge operations in the leveled LSM-structure are counteracting growing search effort, whereas WA is increased and LSM-Trees underperform in all tested settings.

MV-PBT as storage management structure combines beneficial sequential write patterns and almost logarithmic search complexities (compare Section 4.2.4 Equation 4.3) by commonly cached and utilized inner nodes as well as *Cached Partitions*, whereas the number of required merge operations is optimally reduced to GC processes (compare Section 4.4.2). *MV-PBT dominates experimental evaluation of contrastable key-sorted structures. As demonstrated by the last experiment in Figure 6.9, these properties are valid for application of MV-PBT as storage as well as index management structure, leveraging modern in-memory B$^+$-Tree-techniques for the very hot most recent partition.*

6.3. Conclusion

In this chapter, MV-PBT integration details in the prototypical DBMS *PostgreSQL with SIAS* and K/V-Store *WiredTiger 10.0.1* are given and experimentally evaluated by a broad range of modern workload properties. MV-PBT as version-aware index management structure is efficiently able to provide physical references to a snapshot's related tuple versions in SIAS base tables by index-only visibility checks, yielding increased throughput of up to 47% in OLTP workloads and a doubled throughput in analytical processing of HTAP workloads. Nevertheless, in-memory operational maintenance costs might become the bottleneck in very traditional underlying B$^+$-Tree-implementations in the hot fraction of the most recent partition in MV-PBT.

Modern B$^+$-Tree-techniques and in-memory optimizations allow to absorb high update-rates. WT includes very performant B$^+$-Trees and LSM-Trees for in-memory processing and write-intensive tasks respectively. MV-PBT outperforms contrastable storage management structures by up to a factor of 2 for LSM-Trees and orders of magnitude for B$^+$-Trees, whilst sustaining competitive search and scan performance characteristics to B$^+$-Trees. Robust and increased performance characteristics are evaluated for a broad range of applicability, which have also impact on version-aware index management with MV-PBT in DBMS.

MV-PBT exhibits beneficial performance characteristics for a broad range of workload properties, especially in write-intensive workloads, with competitive performance for general purpose. Hence, MV-PBT is qualified as storage and index management structure on modern hardware technologies and recent trends in workload properties.

Chapter 7

Summary and Research Opportunities

This closing chapter briefly summarizes the contributions of this thesis and gives an outlook in research opportunities. MV-PBT enables (index-only) visibility checks in an alpha-numerically sorted structure whilst leveraging modern storage technologies. Research opportunities are settled in the broad range of applicability as well as hardware accelerator technologies, which are probably involved in DBMS [BGHS19] and require to complementary operate on a consistent dataset instance. MV-PBT brings beneficial characteristics for these innovative research areas.

7.1. Summary of Contributions

Multi-Version Partitioned BTrees (MV-PBT) as storage and index management structure considers *tuple versions* as independently maintained and indexed *entities* of one logical tuple – each *valid for a different period* in time. Individually accessible and by partitions locally and temporally separated *version records* of different types are *annotated with timestamp information*, whereas *cheap and robust access latencies* are enabled by *(index-only) visibility checks* – independently from the actual version chain length, since successively performed read I/O with high read amplification (RA) is avoided.

Beneficial strict *out-of-place* as well as *append-only* maintenance of *version records* is firstly introduced in a searchable structure and backed by a *commonly shared MV-PBT-Buffer*, cheap *partition management*, *record types* that indicate different operations and *one-point invalidation* by comprised '*anti matter*'. This *native behavior* in MV-PBT leverages modern Flash secondary storage characteristics, since it massively reduces *write amplification* (WA) and enables a beneficial *sequential write pattern*, whereby MV-PBT is able to cope with amplified effort of version record maintenance.

MV-PBT facilitates recent trends in append-based storage management approaches, e.g. SIAS with *rolling entry points*, by version-aware index management with *low memory footprint*. The potentials of cheaply maintain physical references to massive amounts of individually located tuple version records in base tables eliminates *indirection layers* in main memory and secondary storage, whereby successive operations are combined and resources become available for various purposes.

MV-PBT facilitates a *broad range of applicability* – i.e. principally as a substitution of the ubiquitous B$^+$-Tree, wherein *Partitioned BTrees* [Gra03] and consequently MV-PBT have their origins. Several modern B$^+$-Tree techniques and optimizations, e.g. data skipping and reorganization methods, enable MV-PBT to improve and completely replace B$^+$-Trees as well as LSM-Trees as storage and index management structure – for instance in K/V-Stores like WiredTiger [Mon21] or RocksDB [Inc22].

Immutability of persistent partitions enables beneficial *background online* *adaption* and *reorganization* approaches, like *Cached Partitions* (CP). Thereby, improved caching behavior of commonly used inner nodes facilitate *robust search performance* at high fragmentation. Contrary to LSM-Trees, MV-PBT is not forced to increase write amplification (WA) by necessary merge and rewrite operations of still valid data, however, MV-PBT facilitates *intended garbage collection*, *reorganization* and *defragmentation processes*.

Finally, bloomRF is introduced as *general purpose point-range filter technique*. The lack of *flexible data skipping* structures for *approximate membership testing* necessitate the development of bloomRF in order to introduce MV-PBT as storage and index management structure without constraints and low administrative effort. Various *data type* and *multi-attribute* support with *constant low cost probes* enable *high benefits* for *arbitrary element* and *query interval range span distributions*, which actually *covers the needs in MV-PBT for data skipping*, even if bloomRF's applicability is not limited to.

7.2. Further Application Scenarios of MV-PBT

A broad range of applicability is necessary in widely used storage and index management structures. Application needs vary and rapidly evolve. **RQ2** raised the question for further applications and opportunities of storage and index management with version-aware MV-PBT. A brief overview is given.

7.2.1. Time-Travel Query Processing

In business applications, time-travel queries are an useful capability of DBMS and K/V-Stores [IC20]. Principally, a consistent view on the dataset instance is provided at particular points in time. Implementation of this capability is comparable to MVCC with SI transaction processing (compare Section 2.3), since snapshots provide individual consistent views to transactions. However, the capability of processing time-travel queries massively increases the version chain lengths of logical tuples, since snapshot-related predecessor versions must not be removed. Relevant snapshots are probably defined by *named snapshots* [Mon21], which remain active until they are explicitly dropped. By this means, time-travel queries are enabled for specific points in time, e.g. once a day or month for analytical processing.

According to HTAP query processing in Section 4.3.5, MV-PBT exhibits reasonable characteristics as storage and index management structure in time-travel DBMS and K/V-Stores. First, version-aware MV-PBT is able to perform (index-only) visibility checks (Section 4.3.6). Second, lightweight *minimum transaction timestamps* (Section 4.4.3) allow *lateral entries* to related partition on search operations (Section 4.3.5) and result in robust response times (compare Section 4.3.6.2 Figure 4.21b). Third, discontinuance of logical version chains (Section 4.4.2) allows cheap GC of obsolete version records, which are not related to *named snapshots*. Last, partitions are optionally customized to *named snapshots* by specific GC approaches or creation of sorted views in CP as needed.

7.2.2. Blockchain Storage Structure in Enterprise Systems

Blockchains [Nak09] are a distributed ledger technology, which allows participants to share and maintain data in decentralized networks. Several opportunities rise, for instance in the area of supply chain management [CIS19]. Enterprises are thus able to include the commonly shared and maintained in data in their existing enterprise system landscape [RVP18a] (compare Figure 7.1).

In order to make the shared information valuable in the enterprise system, it is probably necessary to transform blockchain data in a processable format with logical schema and common data types (compare Figure 7.1). This representation enables data integration without affecting the general blockchain load properties. Interestingly, MV-PBT exhibits very beneficial storage and index management characteristics for these properties as well as needs in enterprise systems. The log-based nature of blockchains typically result in append-based storage of immutable information, which is identified by uniformly distributed cryptographic keys. Unspent transactions are rarely formed to an invalid fork of blocks. Nevertheless, off-chain operations and information must be consistently represented and shared in the blockchain

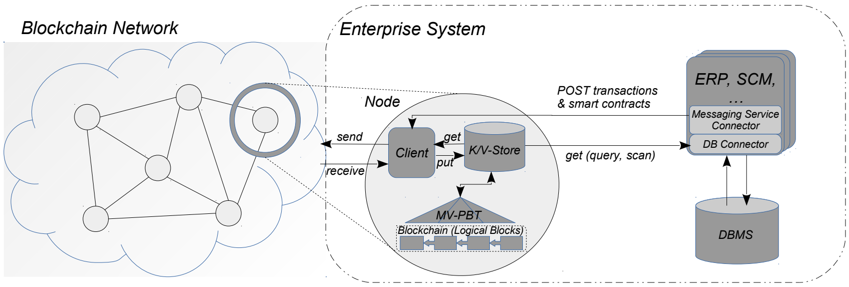


Figure 7.1.: Blockchain Technology included in Enterprise Systems. [RVP18a]

network. MV-PBT leverages append-based storage of new-to-old ordered and processed version tuples, without the need of reorganization for performance reasons (compare Sections 4.2.4 and 4.4.1), similar to information in blockchains. Thereby, MV-PBT enables performant storage and index management characteristics for for high-rate continuous insertion of records with uniformly distributed randomized search keys (compare Section 6.2). Independent partition management of MV-PBTs in a database schema enables separate treatment in storage and index management of off-chain information and operations with a commonly shared MV-PBT-Buffer for very hot and update-intensive partitions (compare Section 4.2). Moreover, data of blockchain forks is cheaply removable by cropping entire partitions (Section 4.4.2). *MV-PBT is an appropriate storage and index management structure for these upcoming kind of workloads.*

7.2.3. Accelerators and Complex Memory Hierarchies

MV-PBT is designed to leverage modern hardware technologies, including highly parallelized and decentralized processing units and accelerators as well as members of the complex memory hierarchy (Sections 2.1.1, 2.1.2, A.1 and A.2). Nevertheless, in implementation and benchmarking (Section

6), only *CPU* as processing unit as well as *RAM* primary storage and *SSD* secondary storage devices are considered.

Partition management yield immutable subsets of data in MV-PBT, which enable several operations, e.g. *online merge sort* (Section 4.3.5), to be commonly processed by individually operating CPUs and accelerators on partitioned data in complex memory hierarchies. Thereby, intelligent cost models probably enable beneficial cooperative execution plans (Section 2.3) for MV-PBT and massively reduce data transfer latencies by smart result set handling for consistent snapshots – e.g. for MVCC with SI, HTAP workloads or time-travel queries – on heterogeneous hardware [KVB+21; VKB+22; VKS+22]. By this means, asynchronous background maintenance operations, e.g. creation of defragmented and consolidated CPs (Section 4.4.1) and GC partitions (Sections 4.4.2), are pushed down to free capacities in accelerators with marginal data movement in the memory hierarchy and effects on payloads.

Besides the capability of native (index-only) visibility checks, MV-PBT benefits from B^+-Tree's increased cache probabilities of sparsely modified inner nodes (compared to horizontally partitioned LSM-Trees, compare Section 4.2.4 and A.3) as well as fully comprised and recoverable meta data in the basic structure. By this means, even search operations in partitions, which comprise *filter probes* (Sections 4.4.3 and Chapter 5), *traversal operations* (compare logarithmic costs in Section 4.2.4) as well as *(index-only) visibility checks* (Section 4.3.6), are commonly processable, based on data locality and available computing power. Especially, lightweight *PBT Cached Meta Data* (Section 4.2.1.1) and *pbt_cursor* information, e.g. *anti_matter_map* (Section 4.2.3), are capable to accelerate and synchronize distributed operations with tiny updates by low latency (near RAM) interfaces for accelerators, as exemplary demonstrated in [BTS+22; TSK+22] for cache-coherent shared lock tables.

MV-PBT is capable to leverage recent developments in hardware technologies, due to applied design decisions, whereby recent trends in workload properties, e.g. HTAP or time-travel queries, are well covered.

Bibliography

[AK21] A. Abdennebi, K. Kaya. *A Bloom Filter Survey: Variants for Different Domain Applications*. 2021. URL: `https://arxiv.org/abs/2106.12189` (cit. on pp. 154, 161).

[AKL13] K. Alexiou, D. Kossmann, P.-Å. Larson. 'Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia'. In: *Proc. VLDB Endow.* 6.14 (Sept. 2013), pp. 1714–1725. URL: `https://doi.org/10.14778/2556549.2556556` (cit. on p. 167).

[APW+08] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, R. Panigrahy. 'Design Tradeoffs for SSD Performance'. In: *USENIX 2008 Annual Technical Conference*. ATC'08. Boston, Massachusetts: USENIX Association, 2008, pp. 57–70 (cit. on p. 30).

[BBB+17] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, A. Lloyd, S. Melnik, R. Rao, D. Shue, C. Taylor, M. van der Holst, D. Woodford. 'Spanner: Becoming a SQL System'. In: *Proc. SIGMOD 2017*. 2017, pp. 331–343 (cit. on p. 37).

[BCKO08] M. d. Berg, O. Cheong, M. v. Kreveld, M. Overmars. *Computational Geometry: Algorithms and Applications*. 3rd ed. Santa Clara, CA, USA: Springer-Verlag TELOS, 2008 (cit. on p. 165).

[BGHS19] P. A. Boncz, G. Graefe, B. He, K.-U. Sattler. 'Database Architectures for Modern Hardware (Dagstuhl Seminar 18251)'. In: *Dagstuhl Reports* 8.6 (2019). Ed. by P. A. Boncz, G. Graefe, B. He, K.-U. Sattler, pp. 63–76. URL: `http://drops.dagstuhl.de/opus/volltexte/2019/10056` (cit. on pp. 28, 231).

[BGO+96] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer. 'An Asymptotically Optimal Multiversion B-tree'. In: *The VLDB Journal* 5.4 (Dec. 1996), pp. 264–275 (cit. on pp. 60, 61).

[BJB09] L. Bouganim, B. Þ. Jónsson, P. Bonnet. 'uFLIP: Understanding Flash IO Patterns'. In: *CoRR* abs/0909.1780 (2009). arXiv: `0909.1780`. URL: `http://arxiv.org/abs/0909.1780` (cit. on pp. 30, 31).

[Blo70] B. H. Bloom. 'Space/Time Trade-Offs in Hash Coding with Allowable Errors'. In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. URL: `https://doi.org/10.1145/362686.362692` (cit. on pp. 9, 91, 162).

[BM03] A. Broder, M. Mitzenmacher. 'Network Applications of Bloom Filters: A Survey'. In: *Internet Mathematics* 1.4 (2003), pp. 485–509. URL: `https://doi.org/` (cit. on pp. 154, 161).

[BM70] R. Bayer, E. McCreight. 'Organization and Maintenance of Large Ordered Indices'. In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET '70. Houston, Texas: Association for Computing Machinery, 1970, pp. 107–141. URL: `https://doi.org/10.1145/1734663.1734671` (cit. on pp. 9, 22, 50, 54, 55).

[BMP+06] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, G. Varghese. 'An Improved Construction for Counting Bloom Filters'. In: *Proceedings of the 14th Conference on Annual European Symposium - Volume 14*. ESA'06. Zurich, Switzerland: Springer-Verlag, 2006, pp. 684–695. URL: `https://doi.org/10.1007/11841036_61` (cit. on p. 161).

[Bos21] Bosch.IO. *Bosch.IO - We bring the IoT to life*. Dec. 2021. URL: `https://bosch.io/` (cit. on p. 19).

[BTS+22] A. Bernhardt, S. Tamimi, F. Stock, T. Vinçon, A. Koch, I. Petrov. 'Cache-Coherent Shared Locking for Transactionally Consistent Updates in Near-Data Processing DBMS on Smart Storage'. In: *EDBT*. OpenProceedings.org, 2022, 2:424–2:428 (cit. on p. 236).

[BU77] R. Bayer, K. Unterauer. 'Prefix B-trees'. In: *ACM Trans. Database Syst.* 2 (1977), pp. 11–26 (cit. on pp. 65, 78).

[Cal19] M. D. Callaghan. 'Diversity of LSM tree shapes'. In: *CIDR*. 2019 (cit. on p. 63).

[CFG+11] R. L. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. A. Kuno, R. O. Nambiar, T. Neumann, M. Pöss, K.-U. Sattler, M. Seibold, E. Simon, F. M. Waas. 'The mixed workload CH-benCHmark'. In: *DBTest '11*. 2011 (cit. on pp. 9, 34, 36, 147, 206, 214).

[CFG+78] L. Carter, R. Floyd, J. Gill, G. Markowsky, M. Wegman. 'Exact and Approximate Membership Testers'. In: *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*. STOC '78. San Diego, California, USA: Association for Computing Machinery, 1978, pp. 59–65. URL: `https://doi.org/10.1145/800133.804332` (cit. on p. 183).

[CIS19] Y. Chang, E. Iakovou, W. Shi. 'Blockchain in Global Supply Chains and Cross Border Trade: A Critical Synthesis of the State-of-the-Art, Challenges and Opportunities'. In: *CoRR* abs/1901.02715 (2019). arXiv: `1901.02715`. URL: `http://arxiv.org/abs/1901.02715` (cit. on p. 234).

[CKZ09] F. Chen, D. A. Koufaty, X. Zhang. 'Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives'. In: *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '09. Seattle, WA, USA: Association for Computing Machinery, 2009, pp. 181–192. URL: `https://doi.org/10.1145/1555349.1555371` (cit. on p. 30).

[CL16] L. Cheng, T. Li. 'Efficient Data Redistribution to Speedup Big Data Analytics in Large Systems'. In: *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. 2016, pp. 91–100 (cit. on p. 19).

[CMB+10] M. Canim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, C. A. Lang. 'SSD Bufferpool Extensions for Database Systems'. In: *Proc. VLDB Endow.* 3.1–2 (Sept. 2010), pp. 1435–1446. URL: `https://doi.org/10.14778/1920841.1921017` (cit. on pp. 161, 162, 173).

[Com79] D. Comer. 'Ubiquitous B-Tree'. In: *ACM Comput. Surv.* 11.2 (June 1979), pp. 121–137. URL: `https://doi.org/10.1145/356770.356776` (cit. on pp. 9, 55).

[CST+10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. 'Benchmarking Cloud Serving Systems with YCSB'. In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154. URL: `https://doi.org/10.1145/1807128.1807152` (cit. on pp. 12, 34–36, 217, 220).

[DAI18a] N. Dayan, M. Athanassoulis, S. Idreos. 'Optimal Bloom Filters and Adaptive Merging for LSM-Trees'. In: *ACM Transactions on Database Systems (TODS)* 43 (2018), pp. 1–48 (cit. on p. 63).

[DAI18b] N. Dayan, M. Athanassoulis, S. Idreos. 'Optimal Bloom Filters and Adaptive Merging for LSM-Trees'. In: *ACM Trans. Database Syst.* 43.4 (Dec. 2018). URL: `https://doi.org/10.1145/3276980` (cit. on p. 161).

[DB20] A. Dholakia, C. Brown. *Data: A Lenovo Solutions Perspective*. 2020. URL: `https://lenovopress.com/lp1367-lenovo-solutions-perspective-on-data` (cit. on p. 35).

[DBp20] DBpedia. *Wikipedia extraction of textual content*. 2020. URL: `https://databus.dbpedia.org/%20dbpedia/text` (cit. on p. 193).

[DFI+13] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, M. Zwilling. 'Hekaton: SQL Server's Memory-Optimized OLTP Engine'. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: Association for Computing Machinery, 2013, pp. 1243–1254. URL: `https://doi.org/10.1145/2463676.2463710` (cit. on pp. 19, 49).

[DHI20] K. Deeds, B. Hentschel, S. Idreos. 'Stacked Filters: Learning to Filter by Structure'. In: *Proc. VLDB Endow.* 14.4 (Dec. 2020), pp. 600–612. URL: `https://doi.org/10.14778/3436905.3436919` (cit. on p. 161).

[DI18] N. Dayan, S. Idreos. 'Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging'. In: *Proceedings of the 2018 International Conference on Management of Data* (2018) (cit. on p. 63).

[DISK20] D. Didona, N. Ioannou, R. Stoica, K. Kourtis. *Toward a Better Understanding and Evaluation of Tree Structures on Flash SSDs*. 2020. arXiv: `2006.04658 [cs.DB]` (cit. on p. 30).

[DM04] P. Dillinger, P. Manolios. 'Bloom Filters in Probabilistic Verification'. In: *Formal Methods in Computer-Aided Design*. Nov. 2004, pp. 367–381 (cit. on pp. 161, 162).

[DR119] S. D. S. S. DR16. *Server data with galaxies, stars and quasars*. 2019. URL: `https://www.kaggle.com/muhakabartay/sloan-digital-sky-survey-dr16` (cit. on p. 194).

[DRZ+16] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, K. Schwan. 'Data Tiering in Heterogeneous Memory Systems'. In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: Association for Computing Machinery, 2016. URL: `https://doi.org/10.1145/2901318.2901344` (cit. on p. 259).

[DSL+11] B. Debnath, S. Sengupta, J. Li, D. J. Lilja, D. H. Du. 'BloomFlash: Bloom Filter on Flash-Based Storage'. In: *2011 31st International Conference on Distributed Computing Systems*. 2011, pp. 635–644 (cit. on pp. 161, 162, 173).

[DT21] N. Dayan, M. Twitto. 'Chucky: A Succinct Cuckoo Filter for LSM-Tree'. In: *Proceedings of the 2021 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 365–378. URL: `https://doi.org/10.1145/3448016.3457273` (cit. on p. 161).

[EGA+18] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Dong, K. Hazelwood, C. Petersen, A. Cidon, S. Katti. 'Reducing DRAM Footprint with NVM in Facebook'. In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: Association for Computing Machinery, 2018. URL: `https://doi.org/10.1145/3190508.3190524` (cit. on pp. 20, 28, 259).

[FAKM14] B. Fan, D. G. Andersen, M. Kaminsky, M. D. Mitzenmacher. 'Cuckoo Filter: Practically Better Than Bloom'. In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and*

Technologies. CoNEXT '14. Sydney, Australia: Association for Computing Machinery, 2014, pp. 75–88. URL: `https://doi.org/10.1145/2674005.2674994` (cit. on pp. 161, 162).

[FCAB00] L. Fan, P. Cao, J. Almeida, A. Z. Broder. 'Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol'. In: *IEEE/ACM Trans. Netw.* 8.3 (June 2000), pp. 281–293. URL: `https://doi.org/10.1109/90.851975` (cit. on p. 161).

[FML+12] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, J. Dees. 'The SAP HANA Database – An Architecture Overview.' In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 28–33. URL: `http://dblp.uni-trier.de/db/journals/debu/debu35.html#FarberMLGMRD12` (cit. on pp. 19, 37).

[FZZ+19] Z. Fan, J. Zhu, Z. Zhang, A. Albarghouthi, P. Koutris, J. M. Patel. 'Scaling-up in-Memory Datalog Processing: Observations and Techniques'. In: *Proc. VLDB Endow.* 12.6 (Feb. 2019), pp. 695–708. URL: `https://doi.org/10.14778/3311880.3311886` (cit. on p. 19).

[Gar15] Gartner. *Real-time Insights and Decision Making using Hybrid Streaming, In-Memory Computing Analytics and Transaction Processing. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation*. Apr. 2015. URL: `https://www.gartner.com/imagesrv/media-products/pdf/Kx/KX-1-3CZ44RH.pdf` (cit. on p. 18).

[GD22] S. Ghemawat, J. Dean. *GitHub - google/leveldb: LevelDB is a fast key-value storage library written at Google that provides an ordered mapping from string keys to string values.* 2022. URL: `https://github.com/google/leveldb` (cit. on pp. 62, 161).

[GGH+14] R. Gottstein, R. Goyal, S. Hardock, I. Petrov, A. Buchmann. 'MV-IDX: Indexing in Multi-Version Databases'. In: *Proceedings of the 18th International Database Engineering & Applications Symposium*. IDEAS '14. Porto, Portugal: Association for Computing Machinery, 2014, pp. 142–148. URL: `https://doi.org/10.1145/2628194.2628911` (cit. on pp. 60, 61, 67).

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995 (cit. on p. 77).

[GJLP14] M. Goswami, A. G. Jørgensen, K. G. Larsen, R. Pagh. 'Approximate Range Emptiness in Constant Time and Optimal Space'. In: *CoRR* abs/1407.2907 (2014). arXiv: 1407.2907. URL: http://arxiv.org/abs/1407.2907 (cit. on p. 183).

[Got16] R. Gottstein. 'Impact of new storage technologies on an OLTP DBMS, its architecture and algorithms.' PhD thesis. Darmstadt: TU, 2016 (cit. on pp. 11, 20, 30, 43, 45, 47, 53, 66, 80, 206, 207, 260).

[GPHB17] R. Gottstein, I. Petrov, S. Hardock, A. P. Buchmann. 'SIAS-Chains: Snapshot Isolation Append Storage Chains'. In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2017, Munich, Germany, September 1, 2017*. Ed. by R. Bordawekar, T. Lahiri. 2017, pp. 50–57. URL: http://www.adms-conf.org/2017/camera-ready/ADMS17%5C_paper%5C_2.pdf (cit. on pp. 44, 45, 53, 66, 67, 117, 208).

[Gra03] G. Graefe. 'Sorting And Indexing With Partitioned B-Trees.' In: *CIDR* (2003) (cit. on pp. 11, 54, 63–65, 68–76, 80, 90, 110, 232).

[Gra04] G. Graefe. 'Write-optimized B-trees'. In: *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 2004, pp. 672–683 (cit. on p. 62).

[Gra11] G. Graefe. 'Modern B-Tree Techniques'. In: *Found. Trends Databases* 3.4 (Apr. 2011), pp. 203–402. URL: https://doi.org/10.1561/1900000028 (cit. on pp. 29, 54, 55, 58, 59, 78, 89, 90, 121).

[GYC+21] G. Gupta, M. Yan, B. Coleman, B. Kille, R. A. L. Elworth, T. Medini, T. Treangen, A. Shrivastava. 'Fast Processing and Querying of 170TB of Genomics Data via a Repeated And Merged BloOm Filter (RAMBO)'. In: *Proceedings of the 2021 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 2226–2234. URL: https://doi.org/10.1145/3448016.3457333 (cit. on p. 161).

[HG20] D. Hieber, G. Grambow. ‘Hybrid Transactional and Analytical Processing Databases: A Systematic Literature Review’. In: *Ninth International Conference on Data Analytics: Held at NexTech 2020*. Oct. 2020, pp. 90–98 (cit. on pp. 18, 34, 35).

[HJF+11] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, S. Zhang. ‘Performance Impact and Interplay of SSD Parallelism through Advanced Commands, Allocation Strategy and Data Granularity’. In: *Proceedings of the International Conference on Supercomputing*. ICS ’11. Tucson, Arizona, USA: Association for Computing Machinery, 2011, pp. 96–107. URL: `https://doi.org/10.1145/1995896.1995912` (cit. on p. 30).

[HSB15] K. Hwang, Y. Shi, X. Bai. ‘Scale-Out vs. Scale-Up Techniques for Cloud Performance and Productivity’. In: *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom* 2015 (Feb. 2015), pp. 763–768 (cit. on p. 19).

[IBM21] IBM. *Internet of Things on IBM Cloud*. Dec. 2021. URL: `https://www.ibm.com/cloud/internet-of-things` (cit. on p. 19).

[IC20] S. Idreos, M. Callaghan. ‘Key-Value Storage Engines’. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2667–2672. URL: `https://doi.org/10.1145/3318464.3383133` (cit. on p. 233).

[Inc22] F. Inc. *GitHub - facebook/rocksdb: A library that provides an embeddable, persistent key-value store for fast storage.* 2022. URL: `https://github.com/facebook/rocksdb` (cit. on pp. 28, 62, 161, 186, 199, 232).

[KBC+18] T. Kraska, A. Beutel, E. H. Chi, J. Dean, N. Polyzotis. ‘The Case for Learned Index Structures’. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ’18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 489–504. URL: `https://doi.org/10.1145/3183713.3196909` (cit. on p. 161).

[KCS+10] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, P. Dubey. ‘FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs’. In: SIGMOD ’10. Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 339–

350. URL: https://doi.org/10.1145/1807167.1807206 (cit. on p. 62).

[KFM+15] M. Kaufmann, P. M. Fischer, N. May, C. Ge, A. K. Goel, D. Kossmann. 'Bi-temporal Timeline Index: A data structure for Processing Queries on bi-temporal data'. In: *2015 IEEE 31st International Conference on Data Engineering*. 2015, pp. 471–482 (cit. on pp. 19, 60).

[KHL18] T. Kissinger, D. Habich, W. Lehner. 'Adaptive Energy-Control for In-Memory Database Systems'. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 351–364. URL: https://doi.org/10.1145/3183713.3183756 (cit. on p. 19).

[KKCL17] Y.-S. Kim, T. Kim, M. J. Carey, C. Li. 'A Comparative Study of Log-Structured Merge-Tree-Based Spatial Indexes for Big Data'. In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 2017, pp. 147–150 (cit. on p. 63).

[KM06] A. Kirsch, M. Mitzenmacher. 'Less Hashing, Same Performance: Building a Better Bloom Filter.' In: *Algorithms–ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings 14*. Springer. Jan. 2006, pp. 456–467 (cit. on p. 161).

[KN11] A. Kemper, T. Neumann. 'HyPer: A Hybrid OLTP and OLAP Main Memory Database System Based on Virtual Memory Snapshots'. In: *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*. ICDE '11. USA: IEEE Computer Society, 2011, pp. 195–206. URL: https://doi.org/10.1109/ICDE.2011.5767867 (cit. on pp. 19, 37).

[Koh18] A. Kohne. 'Service Level Agreements'. In: *Cloud-Föderationen: SLA-basierte VM-Scheduling-Verfahren*. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, pp. 29–34. URL: https://doi.org/10.1007/978-3-658-20973-5_3 (cit. on p. 19).

[KVB+21] C. Knödler, T. Vinçon, A. Bernhardt, I. Petrov, L. Solis-Vasquez, L. Weber, A. Koch. 'A Cost Model for NDP-Aware Query Optimization for KV-Stores'. In: *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*. DAMON'21. Virtual Event, China: Association for Computing Machinery, 2021. URL: https://doi.org/10.1145/3465998.3466013 (cit. on p. 236).

[LC19] C. Luo, M. J. Carey. 'LSM-based storage techniques: a survey'. In: *The VLDB Journal* 29.1 (July 2019), pp. 393–418. URL: https://doi.org/10.1007%2Fs00778-019-00555-y (cit. on pp. 62, 63).

[LCK+20] S. Luo, S. Chatterjee, R. Ketsetsidis, N. Dayan, W. Qin, S. Idreos. 'Rosetta: A Robust Space-Time Optimized Range Filter for Key-Value Stores'. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2071–2086. URL: https://doi.org/10.1145/3318464.3389731 (cit. on pp. 11, 165, 166, 181, 183, 189, 190, 195, 200, 203).

[LDD11] G. Lu, B. Debnath, D. H. Du. 'A Forest-structured Bloom Filter with flash memory'. In: *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*. 2011, pp. 1–6 (cit. on pp. 161, 162, 173).

[LGM+18] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, X. Luo. *Optimizing Bloom Filter: Challenges, Solutions, and Comparisons*. 2018. URL: https://arxiv.org/abs/1804.04777 (cit. on pp. 154, 161).

[LHKN18] V. Leis, M. Haubenschild, A. Kemper, T. Neumann. 'LeanStore: In-Memory Data Management beyond Main Memory'. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. Apr. 2018, pp. 185–196 (cit. on pp. 19, 54).

[LKN13] V. Leis, A. Kemper, T. Neumann. 'The adaptive radix tree: ARTful indexing for main-memory databases'. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE. Apr. 2013, pp. 38–49 (cit. on pp. 19, 62).

[LLS13] J. J. Levandoski, D. B. Lomet, S. Sengupta. 'The Bw-Tree: A B-tree for new hardware platforms'. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 2013, pp. 302–313 (cit. on pp. 9, 19, 41, 45, 62).

[LNKB19] H. Lang, T. Neumann, A. Kemper, P. Boncz. 'Performance-Optimal Filtering: Bloom Overtakes Cuckoo at High Throughput'. In: *Proc. VLDB Endow.* 12.5 (Jan. 2019), pp. 502–515. URL: https://doi.org/10.14778/3303753.3303757 (cit. on pp. 153, 161, 162, 173).

[Lom18] D. Lomet. 'Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed'. In: *Proceedings of the 14th International Workshop on Data Management on New Hardware*. DAMON '18. Houston, Texas: Association for Computing Machinery, 2018. URL: `https://doi.org/10.1145/3211922.3211927` (cit. on p. 20).

[LS90] D. Lomet, B. Salzberg. 'The Performance of a Multiversion Access Method'. In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. SIGMOD '90. Atlantic City, New Jersey, USA: ACM, 1990, pp. 353–363. URL: `http://doi.acm.org/10.1145/93597.98744` (cit. on pp. 60, 61).

[LSP+16] J. Lee, H. Shin, C. G. Park, S. Ko, J. Noh, Y. Chuh, W. Stephan, W.-S. Han. 'Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA'. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 1307–1318. URL: `https://doi.org/10.1145/2882903.2903734` (cit. on pp. 45, 111, 143).

[LZSC20] Q. Liu, L. Zheng, Y. Shen, L. Chen. 'Stable Learned Bloom Filters for Data Streams'. In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 2355–2367. URL: `https://doi.org/10.14778/3407790.3407830` (cit. on p. 161).

[MBL17] N. May, A. Böhm, W. Lehner. 'SAP HANA - The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads'. In: *BTW*. 2017 (cit. on pp. 18, 111).

[MFL14] D. Ma, J. Feng, G. Li. 'A Survey of Address Translation Technologies for Flash Memories'. In: *ACM Comput. Surv.* 46.3 (Jan. 2014). URL: `https://doi.org/10.1145/2512961` (cit. on pp. 31, 32, 261).

[Mic21a] Microsoft. *Azure IoT Hub*. Dec. 2021. URL: `https://azure.microsoft.com/en-us/services/iot-hub/#overview` (cit. on p. 19).

[Mic21b] Microsoft. *SQL Server 2019 | Microsoft*. Dec. 2021. URL: `https://www.microsoft.com/en-US/sql-server/sql-server-2019` (cit. on pp. 37, 41).

[Mit02] M. Mitzenmacher. ‘Compressed Bloom Filters’. In: *IEEE/ACM Trans. Netw.* 10.5 (Oct. 2002), pp. 604–612. URL: https://doi.org/10.1109/TNET.2002.803864 (cit. on p. 161).

[Mit18] M. Mitzenmacher. ‘A Model for Learned Bloom Filters, and Optimizing by Sandwiching’. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS’18. Montréal, Canada: Curran Associates Inc., 2018, pp. 462–471 (cit. on p. 161).

[MKM12] Y. Mao, E. Kohler, R. T. Morris. ‘Cache Craftiness for Fast Multicore Key-Value Storage’. In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys ’12. Bern, Switzerland: Association for Computing Machinery, 2012, pp. 183–196. URL: https://doi.org/10.1145/2168836.2168855 (cit. on p. 62).

[Mon21] MongoDB-Inc. *WiredTiger: WiredTiger Developer Site*. Dec. 2021. URL: https://source.wiredtiger.com/ (cit. on pp. 37, 53, 67, 91, 121, 138, 149, 150, 161, 162, 205, 217–220, 228, 232, 233).

[MRBP23] B. Moessner, C. Riegger, A. Bernhardt, I. Petrov. ‘bloomRF: On Performing Range-Queries in Bloom-Filters with Piecewise-Monotone Hash Functions and Prefix Hashing’. In: *EDBT’23 (accepted)* (2023) (cit. on pp. 9, 24, 164, 168, 175, 176, 183, 184, 186, 191, 192, 194, 195, 199–202).

[MTT00] Y. Manolopoulos, Y. Theodoridis, V. J. Tsotras. *Advanced Database Indexing*. USA: Kluwer Academic Publishers, 2000 (cit. on p. 60).

[MU05] M. Mitzenmacher, E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. USA: Cambridge University Press, 2005 (cit. on p. 183).

[MWKM15] J. Meza, Q. Wu, S. Kumar, O. Mutlu. ‘A Large-Scale Study of Flash Memory Failures in the Field’. In: *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS ’15. Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 177–190. URL: https://doi.org/10.1145/2745844.2745848 (cit. on pp. 20, 28, 32).

[Nak09] S. Nakamoto. ‘Bitcoin: A Peer-to-Peer Electronic Cash System’. In: *Decentralized business review* (May 2009). URL: http://www.bitcoin.org/bitcoin.pdf (cit. on p. 234).

[NAS16] NASA. *Kepler labelled time series exoplanet dataset (Campaign 3)*. 2016. URL: `https://www.kaggle.com/keplersmachines/kepler-labelled-time-series-data` (cit. on p. 192).

[NF20] T. Neumann, M. J. Freitag. 'Umbra: A Disk-Based System with In-Memory Performance'. In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: `http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf` (cit. on p. 20).

[Nuo21] NuoDB. *Multi Version Concurrency Control Part 1: An Overview*. Dec. 2021. URL: `https://nuodb.com/blog/multi-version-concurrency-control-part-1-overview` (cit. on p. 37).

[OCGO96] P. E. O'Neil, E. Cheng, D. Gawlick, E. J. O'Neil. 'The Log-Structured Merge-Tree (LSM-Tree).' In: *Acta Inf.* 33.4 (1996), pp. 351–385. URL: `http://dblp.uni-trier.de/db/journals/acta/acta33.html#ONeilCGO96` (cit. on pp. 10, 62, 64, 80).

[Ora21a] Oracle. *Database 19c and 21c | Oracle*. Dec. 2021. URL: `https://www.oracle.com/database/technologies/` (cit. on p. 37).

[Ora21b] Oracle. *IoT Intelligent Applications*. Dec. 2021. URL: `https://www.oracle.com/internet-of-things/` (cit. on p. 19).

[Ora21c] Oracle. *MySQL :: MySQL 5.6 Reference Manual :: 14.1 Introduction to InnoDB*. Dec. 2021. URL: `https://dev.mysql.com/doc/refman/5.6/en/innodb-introduction.html` (cit. on pp. 37, 41).

[ÖTT17] F. Özcan, Y. Tian, P. Tözün. 'Hybrid Transactional/Analytical Processing: A Survey'. In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 1771–1775. URL: `https://doi.org/10.1145/3035918.3054784` (cit. on pp. 18, 34, 35).

[Pav+21] A. Pavlo et al. *GitHub - oltpbenchmark/oltpbench: Database Benchmarking Framework*. 2021. URL: `https://github.com/oltpbenchmark/oltpbench` (cit. on p. 36).

[Pav14] A. Pavlo. ‘n scalable transaction execution in partitioned main memory database management systems’. Ph.D. dissertation. Brown University, 2014 (cit. on p. 19).

[PCD+21] P. Pandey, A. Conway, J. Durie, M. A. Bender, M. Farach-Colton, R. Johnson. ‘Vector Quotient Filters: Overcoming the Time/Space Trade-Off in Filter Design’. In: *Proceedings of the 2021 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1386–1399. URL: `https://doi.org/10.1145/3448016.3452841` (cit. on p. 161).

[PDZ+18] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, S. Saurabh. ‘Quickstep: A Data Platform Based on the Scaling-up Approach’. In: *Proc. VLDB Endow.* 11.6 (Feb. 2018), pp. 663–676. URL: `https://doi.org/10.14778/3184470.3184471` (cit. on p. 19).

[PKH+19] I. Petrov, A. Koch, S. Hardock, T. Vinçon, C. Riegger. ‘Native Storage Techniques for Data Management’. In: *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 2019, pp. 2048–2051. URL: `https://doi.org/10.1109/ICDE.2019.00236` (cit. on p. 25).

[PKV+19] I. Petrov, A. Koch, T. Vinçon, S. Hardock, C. Riegger. ‘Hardware-Assisted Transaction Processing: NVM’. In: *Encyclopedia of Big Data Technologies*. 2019 (cit. on p. 25).

[Pos21] PostgreSQL. *PostgreSQL: The world’s most advanced open source database*. Dec. 2021. URL: `https://www.postgresql.org/` (cit. on pp. 10, 37, 40, 49, 67, 138, 149, 206, 207).

[PR14] O. Polychroniou, K. A. Ross. ‘Vectorized Bloom Filters for Advanced SIMD Processors’. In: DaMoN ’14. Snowbird, Utah: Association for Computing Machinery, 2014. URL: `https://doi.org/10.1145/2619228.2619234` (cit. on p. 161).

[PSS+10] S.-y. Park, E. Seo, J.-Y. Shin, S. Maeng, J. Lee. ‘Exploiting Internal Parallelism of Flash-based SSDs’. In: *IEEE Computer Architecture Letters* 9.1 (2010), pp. 9–12 (cit. on p. 30).

[PSS10] F. Putze, P. Sanders, J. Singler. ‘Cache-, Hash-, and Space-Efficient Bloom Filters’. In: *ACM J. Exp. Algorithmics* 14 (Jan. 2010). URL: `https://doi.org/10.1145/1498698.1594230` (cit. on pp. 161, 162, 173).

[Pug90] W. Pugh. *Skip Lists: A Probabilistic Alternative to Balanced Trees*. 1990 (cit. on p. 62).

[PVK+19] I. Petrov, T. Vinçon, A. Koch, J. Oppermann, S. Hardock, C. Riegger. ‘Active Storage’. In: *Encyclopedia of Big Data Technologies*. 2019 (cit. on p. 25).

[PWM+14] I. Psaroudakis, F. Wolf, N. May, T. Neumann, A. Böhm, A. Ailamaki, K. Sattler. ‘Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling’. In: *Performance Characterization and Benchmarking. Traditional to Big Data - 6th TPC Technology Conference, TPCTC 2014, Hangzhou, China, September 1-5, 2014. Revised Selected Papers*. Ed. by R. Nambiar, M. Poess. Vol. 8904. Lecture Notes in Computer Science. Springer, 2014, pp. 97–112. URL: `https://doi.org/10.1007/978-3-319-15350-6%5C_7` (cit. on p. 36).

[QCZ+21] Y. Qiao, X. Chen, N. Zheng, J. Li, Y. Liu, T. Zhang. *Closing the B-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression*. 2021. URL: `https://arxiv.org/abs/2107.13987` (cit. on p. 62).

[RBMP20] C. Riegger, A. Bernhardt, B. Moessner, I. Petrov. ‘bloomRF: On Performing Range-Queries with Bloom-Filters based on Piecewise-Monotone Hash Functions and Dyadic Trace-Trees’. In: *CoRR* abs/2012.15596 (2020). arXiv: `2012.15596`. URL: `https://arxiv.org/abs/2012.15596` (cit. on pp. 9, 24, 169, 171, 178).

[RG03] R. Ramakrishnan, J. Gehrke. *Database management systems (3. ed.)* McGraw-Hill, 2003 (cit. on pp. 38, 47, 48, 59, 262).

[RKD21] J. Ren, C. Kjellqvist, L. Deng. *GitHub - basicthinker/YCSB-C: Yahoo! Cloud Serving Benchmark in C++, a C++ version of YCSB*. 2021. URL: `https://github.com/basicthinker/YCSB-C` (cit. on pp. 12, 36, 217, 220).

[RKK12] O. Rottenstreich, Y. Kanizo, I. Keslassy. ‘The Variable-Increment Counting Bloom Filter’. In: *2012 Proceedings IEEE INFOCOM*. 2012, pp. 1880–1888 (cit. on p. 161).

[RMU+14] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, T. Neumann. ‘Locality-sensitive operators for parallel main-memory database clusters’. In: *2014 IEEE 30th International Conference on Data Engineering*. 2014, pp. 592–603 (cit. on p. 19).

[RMVM10] C. E. Rothenberg, C. A. B. Macapuna, F. L. Verdi, M. F. Magalhães. ‘The Deletable Bloom Filter: A New Member of the Bloom Family’. In: *Comm. Letters.* 14.6 (June 2010), pp. 557–559. URL: `https://doi.org/10.1109/LCOMM.2010.06.100344` (cit. on p. 161).

[Röd16] W.-S. Rödiger. ‘Scalable Distributed Query Processing in Parallel Main-Memory Database Systems’. Dissertation. München: Technische Universität München, 2016 (cit. on p. 19).

[RP22] C. Riegger, I. Petrov. ‘Storage Management with Multi-Version Partitioned BTrees’. In: *ADBIS’22 (accepted)* (2022) (cit. on pp. 10, 24, 84, 90, 94, 105, 227).

[RVGP20] C. Riegger, T. Vinçon, R. Gottstein, I. Petrov. ‘MV-PBT: Multi-Version Index for Large Datasets and HTAP Workloads’. In: *Proceedings of the 23rd International Conference on Extending Database Technology (EDBT 2020)*. Copenhagen, Denmark, Mar. 2020. URL: `https://doi.org/10.5441/002/edbt.2020.20` (cit. on pp. 10, 18, 24, 35, 39, 41, 42, 44, 47, 50, 111, 118, 208, 211, 216).

[RVP17a] C. Riegger, T. Vinçon, I. Petrov. ‘Multi-Version Indexing and Modern Hardware Technologies: A Survey of Present Indexing Approaches’. In: *Proceedings of the 19th International Conference on Information Integration and Web-Based Applications & Services*. iiWAS ’17. Salzburg, Austria: Association for Computing Machinery, 2017, pp. 266–275. URL: `https://doi.org/10.1145/3151759.3151779` (cit. on pp. 26, 60).

[RVP17b] C. Riegger, T. Vinçon, I. Petrov. ‘Write-Optimized Indexing with Partitioned B-Trees’. In: *Proceedings of the 19th International Conference on Information Integration and Web-Based Applications & Services*. iiWAS ’17. Salzburg, Austria: Association for Computing Machinery, 2017, pp. 296–300. URL: https://doi.org/10.1145/3151759.3151814 (cit. on pp. 11, 26, 73).

[RVP18a] C. Riegger, T. Vinçon, I. Petrov. 'Efficient Data and Indexing Structure for Blockchains in Enterprise Systems'. In: *Proceedings of the 20th International Conference on Information Integration and Web-Based Applications & Services*. iiWAS2018. Yogyakarta, Indonesia: Association for Computing Machinery, 2018, pp. 173–182. URL: https://doi.org/10.1145/3282373.3282402 (cit. on pp. 25, 234, 235).

[RVP18b] C. Riegger, T. Vinçon, I. Petrov. 'Efficient Data and Indexing Structure for Blockchains in Enterprise Systems'. In: *IBM Technical Report RC25681*. 2018 (cit. on p. 25).

[RVP19] C. Riegger, T. Vinçon, I. Petrov. 'Indexing Large Updatable Datasets in Multi-Version Database Management Systems'. In: *Proceedings of the 23rd International Database Applications & Engineering Symposium*. IDEAS '19. Athens, Greece: Association for Computing Machinery, 2019. URL: https://doi.org/10.1145/3331076.3331118 (cit. on pp. 11, 25, 73).

[RZA+12] X. Ruan, Z. Zong, M. Alghamdi, Y. Tian, X. Jiang, X. Qin. 'Improving write performance by enhancing internal parallelism of Solid State Drives'. In: *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*. Dec. 2012, pp. 266–274 (cit. on p. 30).

[SAP21] SAP. *SAP Business Technology Platform - Intelligent Technologies*. Dec. 2021. URL: https://www.sap.com/products/intelligent-technologies.html (cit. on p. 19).

[SDA21] U. Sirin, S. Dwarkadas, A. Ailamaki. 'Performance Characterization of HTAP Workloads'. In: *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. 2021, pp. 1829–1834 (cit. on p. 18).

[Ser21] A. W. Services. *AWS IoT - Unlock your IoT data and accelerate business growth*. Dec. 2021. URL: https://aws.amazon.com/iot/?nc1=h_ls (cit. on p. 19).

[SG06] A. Szalay, J. Gray. '2020 Computing: Science in an exponential world'. In: *Nature* 440 (Apr. 2006), pp. 413–4 (cit. on p. 35).

[Shi17] I. Shin. 'Verification of performance improvement of multi-plane operation in SSDs'. In: *International Journal of Applied Engineering Research* 12 (Jan. 2017), pp. 7254–7258 (cit. on p. 30).

[SOSM12] F. R. C. Sousa, L. de Oliveira Moreira, G. A. C. Santos, J. C. Machado. 'Quality of Service for Database in the Cloud'. In: *CLOSER*. 2012 (cit. on p. 19).

[SPSA20] S. Sarkar, T. I. Papon, D. Staratzis, M. Athanassoulis. 'Lethe: A Tunable Delete-Aware LSM Engine'. In: SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 893–908. URL: `https://doi.org/10.1145/3318464.3389757` (cit. on p. 80).

[SR12] R. Sears, R. Ramakrishnan. 'BLSM: A General Purpose Log Structured Merge Tree'. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: Association for Computing Machinery, 2012, pp. 217–228. URL: `https://doi.org/10.1145/2213836.2213862` (cit. on pp. 10, 62–64).

[SSZA21] S. Sarkar, D. Staratzis, Z. Zhu, M. Athanassoulis. 'Constructing and Analyzing the LSM Compaction Design Space'. In: *Proc. VLDB Endow.* 14.11 (July 2021), pp. 2216–2229. URL: `https://doi.org/10.14778/3476249.3476274` (cit. on pp. 62, 63, 67).

[ST99] B. Salzberg, V. J. Tsotras. 'Comparison of Access Methods for Time-Evolving Data'. In: *ACM Comput. Surv.* 31.2 (June 1999), pp. 158–221. URL: `https://doi.org/10.1145/319806.319816` (cit. on p. 60).

[Sup21] I. Support. *IBM Support FAQ*. Dec. 2021. URL: `https://www.ibm.com/support/pages/how-much-time-data-stage-etl-job-take-complete` (cit. on p. 19).

[SXX+09] J.-Y. Shin, Z.-L. Xia, N.-Y. Xu, R. Gao, X.-F. Cai, S. Maeng, F.-H. Hsu. 'FTL Design Exploration in Reconfigurable High-Performance SSD for Server Applications'. In: *Proceedings of the 23rd International Conference on Supercomputing*. ICS '09. Yorktown Heights, NY, USA: Association for Computing Machinery, 2009, pp. 338–349. URL: `https://doi.org/10.1145/1542275.1542324` (cit. on p. 30).

[TC21] D. Ting, R. Cole. ‘Conditional Cuckoo Filters’. In: *Proceedings of the 2021 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1838–1850. URL: https://doi.org/10.1145/3448016.3452811 (cit. on p. 161).

[TPC10] TPC. *Transaction Processing Council Benchmark C*. 2010. URL: http://tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf (cit. on pp. 10, 12, 34, 36, 147, 149, 207, 216).

[TPC21] TPC. *Transaction Processing Council Benchmark H*. 2021. URL: http://tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf (cit. on pp. 12, 34).

[TPC22] TPC. *TPC-Homepage*. 2022. URL: http://tpc.org (cit. on pp. 12, 35).

[TRL12] S. Tarkoma, C. E. Rothenberg, E. Lagerspetz. ‘Theory and Practice of Bloom Filters for Distributed Systems’. In: *IEEE Communications Surveys Tutorials* 14.1 (2012), pp. 131–155 (cit. on pp. 154, 161).

[TSK+22] S. Tamimi, F. Stock, A. Koch, A. Bernhardt, I. Petrov. ‘An Evaluation of Using CCIX for Cache-Coherent Host-FPGA Interfacing’. In: *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 2022, pp. 1–9 (cit. on p. 236).

[VHR+18] T. Vinçon, S. Hardock, C. Riegger, J. Oppermann, A. Koch, I. Petrov. ‘NoFTL-KV: TacklingWrite-Amplification on KV-Stores with Native Storage Management’. In: *Proceedings of the 21th International Conference on Extending Database Technology (EDBT 2018)*. Vienna, Austria: OpenProceedings, 2018, pp. 457–460 (cit. on p. 25).

[VHR+19] T. Vinçon, S. Hardock, C. Riegger, A. Koch, I. Petrov. ‘nativeNDP: Processing Big Data Analytics on Native Storage Nodes’. In: *ADBIS*. 2019 (cit. on p. 24).

[VKB+22] T. Vincon, C. Knoedler, A. Bernhardt, L. Solis-Vasquez, L. Weber, A. Koch, I. Petrov. ‘Result-Set Management for NDP Operations on Smart Storage’. In: *Data Management on New Hardware*. DaMoN’22. Philadelphia, PA, USA: Association for Computing Machinery, 2022. URL: https://doi.org/10.1145/3533737.3535097 (cit. on p. 236).

[VKKM20] K. Vaidya, E. Knorr, T. Kraska, M. Mitzenmacher. *Partitioned Learned Bloom Filter*. June 2020 (cit. on p. 161).

[VKS+22] T. Vinçon, C. Knödler, L. Solis-Vasquez, A. Bernhardt, S. Tamimi, L. Weber, F. Stock, A. Koch, I. Petrov. 'Near-Data Processing in Database Systems on Native Computational Storage under HTAP Workloads'. In: *Proc. VLDB Endow.* 15.10 (2022), pp. 1991–2004. URL: `https://www.vldb.org/pvldb/vol15/p1991-petrov.pdf` (cit. on p. 236).

[VWB+20] T. Vinçon, L. Weber, A. Bernhardt, A. Koch, I. Petrov, C. Knödler, S. Hardock, S. Tamimi, C. Riegger. 'nKV in Action: Accelerating KV-Stores on NativeComputational Storage with Near-Data Processing'. In: *Proc. VLDB Endow.* 13.12 (2020), pp. 2981–2984. URL: `http://www.vldb.org/pvldb/vol13/p2981-vincon.pdf` (cit. on p. 24).

[WAL+17] Y. Wu, J. Arulraj, J. Lin, R. Xian, A. Pavlo. 'An Empirical Evaluation of In-Memory Multi-Version Concurrency Control'. In: *Proc. VLDB Endow.* 10.7 (Mar. 2017), pp. 781–792. URL: `https://doi.org/10.14778/3067421.3067427` (cit. on pp. 40, 45).

[WKS15] Y. A. Winata, S. Kim, I. Shin. 'Enhancing internal parallelism of solid-state drives while balancing write loads across dies'. In: *Electronics Letters* 51 (2015), pp. 1978–1980 (cit. on p. 30).

[WLS21] Q. Wang, Y. Lu, J. Shu. *Sherman: A Write-Optimized Distributed B+Tree Index on Disaggregated Memory*. 2021. URL: `https://arxiv.org/abs/2112.07320` (cit. on p. 62).

[WPL+18] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, D. G. Andersen. 'Building a Bw-Tree Takes More Than Just Buzz Words'. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 473–488. URL: `https://doi.org/10.1145/3183713.3196895` (cit. on pp. 9, 41, 45, 62).

[WW21] M. Wong, T. D. Witham. *Database Test Suite*. 2021. URL: `http://osdldbt.sourceforge.net/` (cit. on pp. 36, 206).

[XCJ+17] Z. Xie, Q. Cai, H. Jagadish, B. C. Ooi, W.-F. Wong. ‘Parallelizing Skip Lists for In-Memory Multi-Core Database Systems’. In: *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. 2017, pp. 119–122 (cit. on p. 62).

[XY17] Y. Xia, F. Yang. ‘Locality-based Partitioning for Spark’. In: *2017 5th International Conference on Frontiers of Manufacturing Science and Measuring Technology (FMSMT 2017)*. 2017, pp. 1188–1192 (cit. on p. 19).

[ZBS15] E. Zamanian, C. Binnig, A. Salama. ‘Locality-Aware Partitioning in Parallel Database Systems’. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 17–30. URL: `https://doi.org/10.1145/2723372.2723718` (cit. on p. 19).

[ZCWJ21] W. Zhong, C. Chen, X. Wu, S. Jiang. ‘REMIX: Efficient Range Query for LSM-trees’. In: *FAST*. 2021 (cit. on pp. 19, 63, 166).

[ZLL+18] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, A. Pavlo. ‘SuRF: Practical Range Query Filtering with Fast Succinct Tries’. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ’18. Houston, TX, USA: Association for Computing Machinery, 2018, pp. 323–336. URL: `https://doi.org/10.1145/3183713.3196931` (cit. on pp. 11, 166, 195).

[ZXCX15] J. Zhao, C. Xu, P. Chi, Y. Xie. ‘Memory and Storage System Design with Nonvolatile Memory Technologies’. In: *IPSJ Transactions on System LSI Design Methodology* 8 (Feb. 2015), pp. 2–11 (cit. on p. 259).

[ZZC+17] Y. Zhu, Z. Zhang, P. Cai, W. Qian, A. Zhou. ‘An Efficient Bulk Loading Approach of Secondary Index in Distributed Log-Structured Data Stores’. In: *Database Systems for Advanced Applications - 22nd International Conference, DASFAA 2017, Suzhou, China, March 27-30, 2017, Proceedings, Part I*. Ed. by K. S. Candan, L. Chen, T. B. Pedersen, L. Chang, W. Hua. Vol. 10177. Lecture Notes in Computer Science. Springer, 2017, pp. 87–102. URL: `https://doi.org/10.1007/978-3-319-55753-3%5C_6` (cit. on p. 63).

All links were last followed on August 31, 2022.

Appendix A

Appendix

A.1. Complex Memory Hierarchies

Inside data nodes, data provisioning is performed alongside members of the memory hierarchy. Traditionally, they are differentiated in fast volatile byte-addressable primary, e.g. RAM, and slow persistent block-addressable secondary storage, like disk. These are characterized by symmetric access latencies – i.e. reads and writes perform equally well. Processing units operate on caches and synchronize with working copies in primary storage, which are probably persisted in write I/Os to secondary storage. Non-represent data in primary storage cause read I/O to secondary storage. However, a huge access gap to secondary storage exist.

Semi-conductor storage technologies close this access gap in '*Complex Memory Hierarchies*', however, exhibit special characteristics, like asymmetry, inherent parallelism and out-of-place modifications (compare SSD as an representative in Section A.2).

Based on the underlying technology in NVM, characteristics are varying in between fast byte-addressable RAM-like and asymmetric block-addressable SSD-like behavior [DRZ+16; EGA+18; ZXCX15]. An extensive analysis

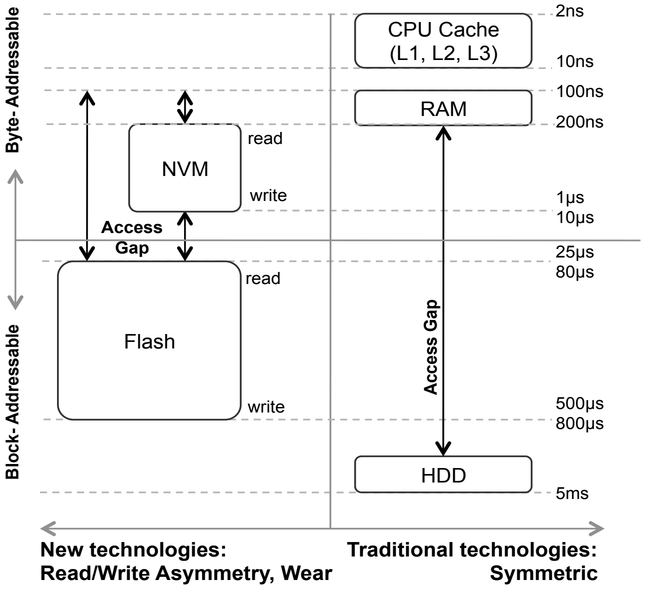


Figure A.1.: Complex Memory Hierarchy in modern Hardware [Got16]

of NVM is out-of-scope in this thesis – generally, characteristics of NVM technologies are settled between RAM and SSD.

A.2. Technical Background: Flash in SSD

In contrast to HDDs, SSDs do not rely on any moving electro-mechanical parts. Data is located in immediate-accessible storage locations without electro-mechanic delay. These fundamental difference is covered by the Flash Translation Layer (FTL) for block-addressable device compatibility, by means that the host system do not take notice of underlying differences, whereas functionality becomes hidden in a black box.

Central task of an FTL is the address translation of logical block addresses (LBA) requested by host to black-boxed physical block addresses (PBA)

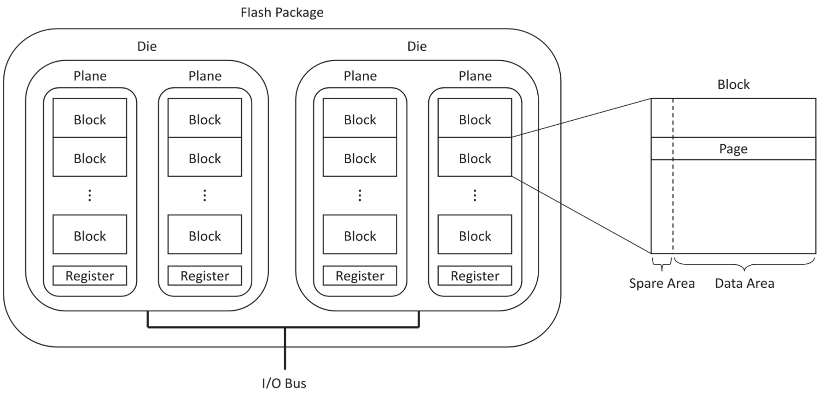


Figure A.2.: Flash package organization. [MFL14]

on device, what is not a static mapping, due to specific behavior of Flash. Several non-volatile Flash memory packages are accessed by Flash Memory Controllers in the FTL. According to [MFL14] and Figure A.2, NAND-based *Flash memory packages* downward subdivide in several mostly independent *dies*, *planes*, *blocks*, *pages* and *memory cells*. NOR-organized Flash memory provide different characteristics and is not in scope in this thesis.

Based on the applied technology, *memory cells* store one (SLC) or multiple bits (MLC/TLC/QLC for 2/3/4 bits per cell) by distinction of load conditions. MLC technology achieves a higher density and therefore larger volume for the sake of performance and endurance. Flash supports *read/write* operations on *page* granularity but *erases* only on *block-level*. Based on this structure and material properties, characteristics of Flash are defined as follows:

- High level of inherent parallelism
- Asymmetric read and write performance
- Out-of-place update operation
- Advantages in sequential write I/O pattern
- Background Garbage Collection
- Limited durability and wear

- I/O transfer time is majority

Details and their relevance to DBMS access structures are outlined in Section 2.1.2.

A.3. Example calculation of single B$^+$-Tree Cache Efficiency

Let a LSM-Tree with 3 components, storing 100, 10 and 1 million leaf nodes in component *C2, C1* and *C0*. Assuming a high fan-out of 200 and a fill factor of 100% – with regards to Equation 3.1, the height of the components is 5, 5 and 4 resulting in 14 individual nodes to search in. Excluding leaf nodes from calculation will result in 11 individual inner nodes.

$$h = \lceil \log_{F \times f_i} \frac{L_N}{f_l} \rceil + 1 \,, \quad F \neq 1 \,, \quad L_N > 1 \,, \; f_i, f_l \in [0.5; 1]$$

$$\begin{aligned} h_{C2} &= \lceil \log_{200\times 1.0} \frac{100E^6}{1.0} \rceil + 1 = 5 \\ h_{C1} &= \lceil \log_{200\times 1.0} \frac{10E^6}{1.0} \rceil + 1 = 5 \\ h_{C0} &= \lceil \log_{200\times 1.0} \frac{1E^6}{1.0} \rceil + 1 = 4 \end{aligned}$$

Maintaining all 111 million leaf nodes in one tree also result in a height of 5, even if the fill factor could be slightly lower than 100%, e.g. 80% in average might be a good assumption in regular redistributed B$^+$-Trees after load [RG03] – a PBT could perform much better as outlined in Section 4.2.4. Searching all partitions requires 15 nodes, however, in this case the root is affected by all three partitions and one more inner node is commonly traversed by the searches of its two smaller partitions[1]. The absolute number of accessed nodes is 12, i.e. 9 individual inner nodes.

[1]Comparing calculations of LSM-components, root nodes of C1 and C0 are very sparely filled, therefore it is very likely that one B$^+$-Tree would combine separator keys in one inner node.

$$h_{one\_tree} = \lceil \log_{200\times 0.8} \frac{111E^6}{0.8} \rceil + 1 = 5$$

Barely 20% *less inner nodes are required in this calculation (individual inner nodes of a B$^+$-Tree* 9 *divided by* 11 *of LSM components). Assuming a p_{ic} of* 0.5 *and p_{lc} of* 0 *in Equation 3.4, LSM requires* 8.5 *read I/Os.*

$$s_{I/O} \approx R_{I/O} \times (\lceil \log_{F\times f_i} \frac{L_N}{f_l} \rceil \times (1 - p_{ic}) + (1 - p_{lc}))\ , \quad p_{ic} \gg p_{lc}$$

$$\begin{aligned}
s_{C2} &\approx R_{I/O} \times (\lceil \log_{200\times 1.0} \frac{100E^6}{1.0} \rceil \times (1 - 0.5) + (1 - 0)) \approx 3.0 R_{I/O} \\
s_{C1} &\approx R_{I/O} \times (\lceil \log_{200\times 1.0} \frac{10E^6}{1.0} \rceil \times (1 - 0.5) + (1 - 0)) \approx 3.0 R_{I/O} \\
s_{C0} &\approx R_{I/O} \times (\lceil \log_{200\times 1.0} \frac{1E^6}{1.0} \rceil \times (1 - 0.5) + (1 - 0)) \approx 2.5 R_{I/O} \\
s_{LSM} &\approx s_{C2} + s_{C1} + s_{C0} \approx 8.5 R_{I/O}
\end{aligned}$$

With 20% *increased cache efficiency (p_{ic} =* 0.6*) of inner nodes and three partitions (P), a single tree requires only* 7.8 *random read I/Os.* Compared to 8.5 read I/O of LSM-Trees, searching in the horizontally partitioned B$^+$-Tree requires about 8% less read I/Os from significantly slower secondary storage.

In this calculations, fixed assignment of *C0* to main memory is not considered, since this functionality shrinks cache efficiency of further components. Furthermore, this behavior should also be applied in Partitioned B-Trees for comparability.

$$\begin{aligned}
s_{one\_tree} &\approx R_{I/O} \times P \times ((\lceil \log_{F\times f_i} \frac{L_N}{f_l} \rceil) \times (1 - p_{ic}) + (1 - p_{lc})) \\
&\approx R_{I/O} \times 3 \times ((4) \times (1 - 0.6) + (1)) \\
&\approx 7.8 R_{I/O}
\end{aligned}$$