Solid-State Storage and Work Sharing

for

Efficient Scaleup Data Analytics

Thèse n. 6032 (2013)
présenté le 12 Décembre 2013
à la Faculté Informatique et Communications
laboratoire de systèmes et applications de traitement de données massives
programme doctoral en Informatique et Communications
École Polytechnique Fédérale de Lausanne

pour l'obtention du grade de Docteur ès Sciences
par

Manoussos-Gavriil Athanassoulis

acceptée sur proposition du jury:

Prof. George Candea, président du jury
Prof. Anastasia Ailamaki, directeur de thèse
Prof. Willy Zwaenepoel, rapporteur
Prof. Kenneth A. Ross, rapporteur
Dr Phillip B. Gibbons, rapporteur

Lausanne, EPFL, 2013

I was never more certain of
how far away I was from my goal
than when I was standing right beside it.

To my family, to my teachers, to my friends.
All of them teachers in so many ways …

Acknowledgements

The author of this book is only one, however, without the support of a number of people it would not have been possible to produce this work. Many people deserve recognition both for their technical and their general support which helped me complete this thesis. If I missed anyone below, it is because I was lucky to have many such people in my life and unlucky to have a lousy memory.

My advisor Natassa Ailamaki, reignited the research bug when she visited the University of Athens and gave an inspiring talk, at the time that I was about to complete my Master studies and contemplating for my future. Soon thereafter I joined her lab, DIAS, at EPFL where she taught me nearly everything about database and systems research. Natassa is an enthusiastic advisor and researcher - one could say a research evangelist - who can give technical and motivational advices when most needed.

I would also like to thank Shimin Chen and Phil Gibbons for our excellent collaboration. Their research rigor and focus on the detail was a great lesson to me.

Phil Gibbons also participated in my PhD committee, and along with him I thank Willy Zwaenepoel, Ken Ross, and George Candea for agreeing to be part of my PhD committee.

The members of the DIAS lab were a big part of my PhD journey. Following chronological order, I would like to thank Ryan Johnson, Radu Stoica, Ippokratis Pandis, Ioannis Alagiannis, Iraklis Psaroudakis, and Stratos Idreos for our fruitful collaboration during my PhD years. Ryan Johnson was my first office-mate, and in fact a perfect one, dedicated to high quality research and eager to offer any help I ever needed. Ioannis Alagiannis, a lifelong friend, is just the person that is most natural to collaborate with. In addition to these people, I would like to thank each and every DIAS lab member for creating a healthy work environment I was happy to go to every day during my PhD years. Verena Kantere, Debabrata Dash, Adrian Popescu, Renata Borovica, Danica Porobic, Pınar Tözün, Farhan Tauheed, Thomas Heinis, Erietta Liarou, Miguel Branco, Manos Karpathiotakis, Mirjana Pavlovic, Eleni Tzirita-Zacharatou, and Matthaios Olma offered, as well, invaluable discussions and comments for my ongoing research and my presentations.

I also spent four great months in IBM Watson Research Center in New York, USA. For this great experience I would like to thank Bishwaranjan Bhattacharjee, Ken Ross, Mustafa Canim, and Yuan-chi Chang.

The bug of research has been plugged into me since I was a child, having endless discussions with the first researcher I ever met, my father. His passion for research was inspiring for me: before the age of ten I announced to my parents, Makis and Eleni, that I want to be an "experimentalist". My parents gave all the help I ever needed to study what I liked most and gave me the bug to seek for answers of interesting questions. My older brother, Agis, who follows the academic path as well, was always giving to me a first-hand taste of what this path looks like, and offered – during both undergraduate and graduate studies – invaluable help and advice.

I was lucky to complete my undergraduate studies in an excellent school, the Department of Informatics and Telecommunications of University of Athens, Greece. During my studies I had inspiring teachers including Alex Delis who exemplifies research excellence and integrity, and Stathes Hadjiefthymiades who supervised both my undergraduate and masters thesis, introducing me for real into the research world.

I greatly appreciate the effort made by Pierre Grydbeck and Dimitra Tsaoussis to help me write the French abstract of my thesis.

Finally, I gratefully acknowledge the funding supporting my PhD research, as part of my IBM PhD Fellowship Award, from the Swiss National Science Foundation EURYI Award, Grant No PE0022-117125/1, and from the European Commission Collaborative Project “BigFoot”, Grant Agreement Number 317858.

Lausanne, 12 December 2013 M. A.

Abstract

Today, managing, storing and analyzing data continuously in order to gain additional insight is becoming commonplace. Data analytics engines have been traditionally optimized for read-only queries assuming that the main data reside on mechanical disks. The need for 24x7 operations in global markets and the rise of online and other quickly-reacting businesses make *data freshness* an additional design goal. Moreover, the increased requirements in information quality make *semantic databases* a key (often represented as graphs using the RDF data representation model). Last but not least, the performance requirements combined with the increasing amount of stored and managed data call for *high-performance yet space-efficient* access methods in order to support the desired concurrency and throughput.

Innovative data management algorithms and careful use of the underlying hardware platform help us to address the aforementioned requirements. The volume of generated, stored and queried data is increasing exponentially, and new workloads often are comprised of time-generated data. At the same time the hardware is evolving with dramatic changes both in processing units and storage devices, where solid-state storage is becoming ubiquitous. In this thesis, we build workload-aware data access methods for data analytics - tailored for emerging time-generated workloads - which use solid-state storage, either (i) as an additional level in the memory hierarchy to enable real-time updates in a data analytics, or (ii) as standalone storage for applications involving support for knowledge-based data, and support for efficiently indexing archival and time-generated data.

Building workload-aware and hardware-aware data management systems allows to increase their performance and to *augment their functionality*. The advancements in storage have led to a variety of storage devices with different characteristics (e.g., monetary cost, access times, durability, endurance, read performance vs. write performance), and the suitability of a method to an application depends on how it balances the different characteristics of the storage medium it uses. The data access methods proposed in this thesis - MaSM and BF-Tree - balance the benefits of solid-state storage and of traditional hard disks, and are suitable for time-generated data or datasets with similar organization, which include social, monitoring and archival applications. The study of work sharing in the context of data analytics paves the way to integrating shared database operators starting from shared scans to several data analytics engines, and the workload-aware physical data organization proposed for knowledge-based datasets - RDF-tuple - enables integration of diverse data sources into the same systems.

Keywords: databases, data analytics, solid-state storage, data freshness, work sharing, access methods

Résumé

De nos jours, la gestion, le stockage et l'analyse de données en continu, afin d'obtenir des informations supplémentaires, devient courant. Les systèmes d'analyse de données ont été traditionnellement optimisés pour des requêtes en lecture seule en supposant que les données principales résidaient sur des disques mécaniques. Le besoin d'opérer 24 heures sur 24, 7 jours sur 7 dans le contexte de marchés globaux et suite à l'accroissement des activités commerciales en ligne ont fait de la « fraîcheur des données » un objectif supplémentaire. En outre, les exigences accrues en matière de qualité de l'information rendent les bases de données sémantiques clés (souvent représentées comme des graphes utilisant le modèle de représentation de données RDF). Finalement, les contraintes de performances combinées avec l'augmentation du volume des données stockées et gérées nécessitent des méthodes d'accès performantes mais efficaces en termes de stockage afin de garantir le niveau de concurrence et de débit souhaité.

Des algorithmes de gestion de données innovants et l'utilisation attentive de la plateforme matérielle sous-jacente nous aident à répondre aux exigences citées. Le volume des données générées, stockées et demandées augmente exponentiellement et les nouvelles charges de travail contiennent souvent des données temporelles. En même temps, le matériel informatique évolue avec des changements drastiques autant dans les unités de traitement que dans le stockage où les lecteurs à état solide deviennent omniprésents. Dans cette thèse, nous construisons des méthodes d'accès s'adaptant aux charges de travail d'analyse de données. Elles sont faites sur mesure pour les données temporelles qui utilisent le stockage à l'état solide soit (i) comme un niveau supplémentaire dans la hiérarchie de stockage permettant des mises à jour en temps réel des analyses de données ou (ii) comme stockage indépendant pour des applications nécessitant le support de données basées sur la connaissance et le soutient pour l'indexation efficace de données d'archive.

Les systèmes de gestion de données utilisant la connaissance de la charge de travail et de la plateforme matérielle sous-jacente peuvent augmenter leurs performances et fonctionnalités. L'avance dans le domaine des périphériques de stockage a mené au développement d'une variété de caractéristiques différentes (par exemple : coûts monétaires, temps d'accès, durabilité, endurance, performance de lecture contre écriture, etc.). L'efficacité d'une méthode d'accès pour une application donnée dépend de comment ces caractéristiques sont conciliées. Les méthodes d'accès proposées dans cette thèse, MaSM et BF-Tree, équilibrent les gains du stockage à l'état solide avec le stockage sur disques durs traditionnels et sont appropriées pour les données temporelles ou avec une structure similaire. Ceci comprend les applications sociales, de contrôle ou d’archivage. L’étude du partage du travail dans le contexte des analyses de données ouvre la voie à l’intégration d’opérateurs de bases de données partagés à partir de scans communs à plusieurs systèmes d’analyse de données et avec l’organisation physique des données basée sur les charges de travail et les ensembles de données liées à des connaissances (RDF-tuple) permettent l’intégration de plusieurs sources de données dans un même système.

Mots-clés : bases de données, analyse de données, stockage à l’état solide, fraîcheur des données, partage du travail, méthodes d’accès

Contents

Acknowledgements **v**

Abstract (English/Français) **vii**

Table of Contents **xv**

List of figures **xix**

List of tables **xxi**

1 Introduction **1**
- 1.1 The Information Age 1
- 1.2 Data Management 1
- 1.3 Data Analytics 3
  - 1.3.1 Data Analytics Challenges 4
- 1.4 Implications on the DBMS architecture 4
- 1.5 Solid-State Storage and Work Sharing for Efficient Scaleup Data Analytics 5
  - 1.5.1 Evolution of the storage hierarchy 6
  - 1.5.2 Summary and Contributions 7
  - 1.5.3 Published papers 8
- 1.6 Outline (How to read this thesis) 9

2 Related Work and Background **11**
- 2.1 Hardware Trends in Storage 11
  - 2.1.1 Fifty years of hard disks 11
  - 2.1.2 From hard disks to solid-state storage 12
  - 2.1.3 Flash-based solid-state storage 13
  - 2.1.4 Phase Change Memory 14
  - 2.1.5 Flash vs. PCM 15
  - 2.1.6 More solid-state storage technologies 17
  - 2.1.7 Sequential Reads and Random Writes in Storage Systems 18
- 2.2 Data Analysis Systems 18
  - 2.2.1 High-Throughput Updates and Data Analysis 19
  - 2.2.2 Adaptive organization of data and updates 20

2.2.3 Orthogonal Uses of SSDs for Data Warehouses 21
2.3 Optimizing Data Analysis Queries Using Work-Sharing 21
2.3.1 Sharing in the I/O layer and in the execution engine 22
2.3.2 Simultaneous Pipelining 23
2.3.3 The QPipe execution engine 23
2.3.4 Global Query Plans with shared operators 24
2.3.5 The CJOIN operator 25
2.3.6 Quantifying work sharing opportunities 27
2.4 Bloom filters 28
2.4.1 Bloom filters' applications in data management 28
2.4.2 Bloom filters for evolving workloads and storage 28
2.5 Indexing for Data Analysis 29
2.5.1 SSD-aware indexing 29
2.5.2 Specialized optimizations for tree indexing 29
2.5.3 Indexing for data warehousing 30
2.6 Knowledge-based Data Analysis 30
2.6.1 RDF datasets and benchmarks 30
2.6.2 Storing RDF data 31
2.7 Summary 32

3 Online Updates for Data Analytics using Solid-State Storage 33
3.1 Introduction 33
3.1.1 Efficient Online Updates for DW: Limitations of Prior Approaches 34
3.1.2 Our Solution: Cache Updates in SSDs for online updates in DWs 35
3.1.3 Contributions 38
3.2 Efficient Online Updates and Range Scans in Data Warehouses 38
3.2.1 Basic Concepts and Focus of the Study 39
3.2.2 Conventional Approach: In-Place Updates 40
3.2.3 Prior Proposals: Indexed Updates (IU) 41
3.3 MaSM Design 42
3.3.1 Basic Ideas 43
3.3.2 MaSM-2M 44
3.3.3 MaSM-M 46
3.3.4 MaSM-αM 49
3.3.5 Further Optimizations 49
3.3.6 Transaction Support 50
3.3.7 Achieving The Five Design Goals 52
3.4 Analysis and Modeling of MaSM 53
3.4.1 Theorems About MaSM Behavior 54
3.4.2 SSD Wear vs. Memory Footprint 55
3.4.3 Memory Footprint vs. Performance 56
3.4.4 Modeling the I/O Time of Range Queries 58

3.4.5 LSM analysis and comparison with MaSM 62
3.5 Experimental Evaluation 64
3.5.1 Experimental Setup 64
3.5.2 Experiments with Synthetic Data 65
3.5.3 TPC-H Replay Experiments 71
3.6 Discussion 71
3.6.1 Shared-Nothing Architectures 71
3.6.2 MaSM for deeper memory hierarchies and new storage technologies . . . 72
3.6.3 Can SSDs Enhance Performance and Be Cost Efficient? 73
3.6.4 General support of MaSM 73
3.6.5 Applying MaSM to Cloud Data Management 75
3.7 Conclusion 75

4 Enhancing Data Analytics with Work Sharing 77
4.1 Introduction 77
4.1.1 Methodologies for sharing data and work 77
4.1.2 Integrating Simultaneous Pipelining and Global Query Plans 78
4.1.3 Optimizing Simultaneous Pipelining 79
4.1.4 Simultaneous Pipelining vs. Global Query Plans 79
4.1.5 Contributions 80
4.1.6 Outline 81
4.2 Integrating SP and GQP 81
4.2.1 Benefits of applying SP to GQP 81
4.2.2 CJOIN as a QPipe stage 82
4.2.3 SP for the CJOIN stage 83
4.3 Shared Pages Lists for SP 84
4.3.1 Design of a SPL 87
4.3.2 Linear Window of Opportunity 87
4.4 Experimental Methodology 88
4.5 Experimental Analysis 89
4.5.1 Impact of concurrency 90
4.5.2 Impact of data size 92
4.5.3 Impact of Similarity 95
4.5.4 SSB query mix evaluation 96
4.6 Discussion 97
4.7 Conclusions 98

5 BF-Tree: Approximate Tree Indexing 101
5.1 Introduction 101
5.1.1 Implicit Clustering 101
5.1.2 The capacity-performance trade-off 102
5.1.3 Indexing for modern storage 103
5.1.4 Approximate Tree Indexing 103

5.2 Making approximate indexing competitive . . . 105
5.3 Bloom Filter Tree (BF-Tree) . . . 106
  5.3.1 BF-Tree architecture . . . 106
  5.3.2 Creating and Updating a BF-Tree . . . 107
5.4 Read-optimized BF-Trees . . . 110
5.5 Modeling BF-Trees and B$^+$-Trees . . . 111
  5.5.1 Parameters and model . . . 111
  5.5.2 Discussion on BF-Tree size and compression . . . 115
5.6 Experimental evaluation . . . 116
  5.6.1 Experimental Methodology . . . 116
  5.6.2 BF-Tree for primary key . . . 117
  5.6.3 BF-Tree for non-unique attributes . . . 121
  5.6.4 BF-Tree for TPCH . . . 123
  5.6.5 Summary . . . 125
5.7 BF-Tree as a general index . . . 125
  5.7.1 BF-Tree vs. interpolation search . . . 125
  5.7.2 Range scans . . . 126
  5.7.3 Querying in the presence of inserts and deletes . . . 126
5.8 Optimizations . . . 129
5.9 Conclusions . . . 129

6 Graph Data Management using Solid-State Storage 131
6.1 Introduction . . . 131
  6.1.1 Contributions . . . 132
  6.1.2 Outline . . . 133
6.2 Path Processing . . . 133
6.3 RDF processing on PCM . . . 137
  6.3.1 The RDF-tuple . . . 139
  6.3.2 The Internals of Pythia . . . 140
  6.3.3 Experimental Workload . . . 141
  6.3.4 Experimental Evaluation . . . 142
  6.3.5 Comparison with RDF-3X . . . 143
6.4 PCM Opportunities . . . 144
  6.4.1 Algorithm redesign . . . 145
6.5 SPARQL code for test queries . . . 145
6.6 Conclusions . . . 146

7 The Big Picture 149
7.1 What we did . . . 149
7.2 Storage is constantly evolving: Opportunities and challenges . . . 150
  7.2.1 Persistent main memory: Database Management Systems vs. Operating Systems . . . 151
  7.2.2 Software stack is too slow . . . 151

7.3 Ever increasing concurrency in analytics 151
7.3.1 Extending support of work sharing 152
7.3.2 Column-stores and high concurrency 152

Bibliography **153**

Curriculum Vitae **167**

List of Figures

2.1 Fifty years of HDD 12
2.2 Read latency for (a) flash (zoomed in the most relevant part) and (b) PCM 16
2.3 Write latency for (a) flash (zoomed in the most relevant part) and (b) PCM . . . 17
2.4 (a) SP example with two queries having a common sub-plan below the join operator. (b) A step and a linear WoP. 23
2.5 Example of shared selection and hash-join operators. 24
2.6 CJOIN evaluates a GQP for star queries. 26

3.1 An analysis of migration overheads for differential updates as a function of the memory buffer size. Overhead is normalized to the prior state-of-the-art using 16GB memory. 34
3.2 Framework for SSD-based differential updates. 37
3.3 TPC-H queries with random updates on a row store. 40
3.4 TPC-H queries with emulated random updates on a column store. 40
3.5 Extending prior proposals of Indexed Updates (IU) to SSDs: (a) in-memory indexed updates (PDT (Héman et al., 2010)), (b) extending IU to SSDs, and (c) applying LSM (O'Neil et al., 1996) to IU. 41
3.6 Illustrating the MaSM algorithm using $2M$ memory. 44
3.7 MaSM-2M algorithm 45
3.8 Illustrating MaSM algorithm using M memory. 47
3.9 MaSM-M algorithm 48
3.10 The tradeoff between SSD wear and memory footprint. 56
3.11 Memory footprint vs. short range performance. 57
3.12 Modeling MaSM performance for reading main data and fetching the updates using the characteristics of the storage used in our experimentation. 61
3.13 The ratio of the time to fetch the updates from SSD to the time to read the main data for index granularity 64KB. 61
3.14 The ratio of the time to fetch the updates from SSD to the time to read the main data for index granularity 4KB. 61
3.15 Modeling MaSM performance for reading main data and fetching the updates using the characteristics of a high-end storage system. 62
3.16 Comparing the impact of online update schemes on the response time of primary key range scans (normalized to scans without updates). 66

3.17 MaSM range scans varying updates cached in SSD. 67
3.18 Range scan with MaSM varying update speed. 67
3.19 Comparing the impact of online update schemes, with MaSM using both SSD and HDD, on the response time of primary key range scans (normalized to scans without updates). 68
3.20 MaSM update migration. 68
3.21 Sustained updates per second varying SSD size in MaSM. 68
3.22 Range scan and MaSM performance while emulating CPU cost of query processing (10GB ranges). 69
3.23 Comparing in-place updates with MaSM performance (using both coarse-grain and fain-grain run indexes) when both data and updates are stored in memory (10GB table size). 70
3.24 Replaying I/O traces of TPC-H queries on a real machine with online updates. . . 71

4.1 Evaluation of three concurrent queries using (a) a query-centric model, (b) shared scans, (c) SP, and (d) a GQP. 78
4.2 Integration scheme of CJOIN as a QPipe stage. 83
4.3 Evaluating multiple identical TPC-H Q1 queries (a) with a push-only model during SP (FIFO), and (b) with a pull-based model during SP (SPL). In (c), we show the corresponding speedups of the two methods of SP, over not sharing, for low concurrency. 84
4.4 Sharing identical results during SP with: (a) push-only model and (b) a SPL. . . . 85
4.5 Design of a shared pages list. 86
4.6 Same experiment as in Figure 4.3b, for the case of 8 concurrent queries, with a varying maximum size for SPL. 87
4.7 The SSB Q3.2 SQL template and the query plan. 90
4.8 Memory-resident database of SF=1. The table includes measurements for the case of 256 queries. 90
4.9 Same as Figure 4.8, with a disk-resident database. 91
4.10 8 queries with a memory-resident database of SF=10. The table includes measurements for 30% selectivity. 92
4.11 Memory-resident database of SF=10 and 30% selectivity. The table includes measurements for 256 queries. 93
4.12 8 concurrent queries with disk-resident databases. The table includes measurements for the case of SF=100. 94
4.13 Disk-resident database of SF=1 and 16 possible plans. The table includes measurements for 256 queries. 95
4.14 Evaluating 512 concurrent queries with a varying similarity factor, for a SF=100. The table includes the SP sharing opportunities (average of all iterations), in the format 1st/2nd/3rd hash-join for QPipe-SP. 96
4.15 Disk-resident database of SF=30. The table includes measurements for the case of 256 queries. 97

4.16 Throughput while evaluating concurrent queries with a disk-resident database of SF=30. The table includes measurements for the case of 256 clients. 98

5.1 Clustering of shipdate, commitdate, and receiptdate. 102
5.2 The capacity/performance storage trade-off. 104
5.3 BF-tree 107
5.4 BF-tree leaves 110
5.5 Analytical comparison of BF-Tree vs. B^+-Tree. 115
5.6 BF-Tree and B^+-Tree performance for the PK index for five storage configurations for storing the index and the main data. 117
5.7 The break-even points when indexing PK 119
5.8 BF-Tree and B^+-Tree performance for PK index: cold caches vs. warm caches. . 120
5.9 BF-Tree and B^+-Tree performance for ATT1 index for five storage configurations for storing the index and the main data. 121
5.10 The break-even points when indexing ATT1 122
5.11 BF-Tree and B^+-Tree performance for ATT1 index: cold caches vs. warm caches. 123
5.12 BF-Tree for point queries on TPCH dates varying hit rate. 124
5.13 IOs for range scans using BF-Trees vs. B^+-Trees. 126
5.14 False positive probability (fpp) in the presence of inserts. 128
5.15 False positive probability (fpp) in the presence of inserts (zoomed in). 128

6.1 Latency for 4K, 8K, 16K, 32K page size when varying concurrent queries. Note the logarithmic scale on both axes. 135
6.2 Bandwidth for 4K, 8K, 16K, 32K page size when varying concurrent queries . . 136
6.3 Custom path processing benchmark: Speedup of PCM over flash 137
6.4 Latency per page request size for PCM and flash, varying page size and number of concurrent threads issuing requests. 138
6.5 Bandwidth for PCM and flash, varying page size and number of concurrent threads issuing requests. 138
6.6 RDF-tuple in the SPO-store 139
6.7 RDF-tuple layout 140
6.8 Pythia Architecture 141
6.9 Bandwidth for Pythia for 1, 2, 4, 8, 16, 32 concurrent clients submitting queries . 143
6.10 Latency for Pythia for 1, 2, 4, 8, 16, 32 concurrent clients submitting queries . . . 143
6.11 Query latency using Pythia for Q1, Q2 and Q3 144

List of Tables

2.1 HDD trends compared and flash characteristics 13
2.2 Sharing methodologies employed by query-centric models and the research prototypes we examine. 21

3.1 Parameters used in the MaSM-M algorithm. 49
3.2 The parameters of MaSM-M's performance model and their values. 59
3.3 Computing the optimal height, h for an LSM tree, using the size ratio between adjacent levels, r, and the I/O write rate normalized by input insertion rate, $(h-0.5)(1+r)$. (The C_0 tree is in memory, and C_1, ..., C_h trees are on SSDs. Suppose memory size allocated to the C_0 tree is 16MB, and the SSD space allocated to the LSM tree is 4GB.) 62

4.1 Rules of thumb for when and how to share data and work across typical concurrent analytical queries in DW. 80

5.1 Parameters of the model 112
5.2 B$^+$-Tree & BF-Tree size (pages) for 1GB relation 118
5.3 False reads/search for the experiments with 1GB data 118

6.1 Description of the benchmark 133
6.2 Hardware specification of the PCM device 134
6.3 Hardware specification of the flash device 134
6.4 Custom graph benchmark parameters. 134

1 Introduction

1.1 The Information Age

The saying *"Knowledge is power"*[1], lays the groundwork for how the era we live in, the Information Age, is affecting humanity. The Information Age is characterized by the growing need to gather, manage, access and analyze information to support the operation of any commercial, research or government organization. It is based upon the digital revolution - also known as the third industrial revolution. In turn the digital revolution is based on the enormous and continuous advancements in technology which allows us to create digital circuits capable of storing, processing, exchanging, analyzing and generating exponentially more digital data every year.

Information and data management, however, is not a recent discipline. Centuries before the digitalization of data, businesses and governments used centralized data repositories to keep financial, managerial and organizational information. Such applications required functionality guarantees; namely, correctness, consistency and permanence of the information. The growing need to store and manage data combined with the technological advancements led to the development of data management systems as a result of both research efforts and commercial applications.

1.2 Data Management

The need to store data in order to facilitate governmental organization and business transactions was met when technology started offering ways to automate such processes. In fact, this need fueled the explosion of research and development in computer-based systems to centrally maintain bank, business and governmental information which started in the 1950s when the notion of electronic computers became reality. In a few years time a new market had been created and already by the 1960s companies were working on building complex software systems on top of expensive computers to store, manage and query data: the data managements systems market had been created.

---

[1] "Scientia potentia est" in Latin, attributed to Sir Francis Bacon

One of the first widely used database management system was IBM's IMS (Information Management System), initially designed for the 1966 Apollo Program. IMS used a hierarchical conceptual model where data were organized in a tree-structure which is capable of representing parent-child relationships: one parent may have many children, but each child has one parent. Apart from supporting travel in space, IMS was widely used in the banking sector supporting every-day transactions by banks and bank customers. In 1970s Edgar Codd introduced the Relational Model (Codd, 1970) which shaped how the majority of the systems to follow represent and store data. The Relational Model, which is based on first-order logic, represents data using *tuples* which, when grouped, form *relations*. In the Relational Model the logical organization of the data (tuples and relations) is decoupled from the physical organization of the data. This flexibility and the clarity of representing information lead to a wide adoption of the Relational Model which became the standard way to represent information in database systems.

Following the development of the Relational Model, several relational database management systems (RDBMS) were developed either as research prototypes or as commercial products. The evolution of the database market led to the development of a query language which today is known as SQL. While the initial domain of application for database systems was mostly the governmental and the banking applications, the development of the Internet since the mid-1990s caused an exponential growth of the database market. The ease in connectivity led to the creation of numerous client-server applications that required database support, and at the same time, more and more data were generated. The trend of increasing data generation and, consequently, the need to store it and manage it, is still growing at exponential rates.

Soon after the first simple web applications, the data generated by users of online services, by customers of online and offline shops, and in general by any user of any digitalized service was stored to ensure the correct operation of all these businesses and services. At the same time, archived data can reveal additional information if one has the means to analyze it efficiently. Starting from the late 1990s an increasing number of database systems vendors started focusing on building systems to efficiently manage and analyze enormous amounts of data. Initially, database management systems vendors needed to address the requirements of two major application genres. First, to keep safely the accurate and up-to-date information of transactions between businesses and people leading to the development of On-line Transactional Processing (OLTP). Second, to manage and analyze huge and increasing quantities of data resulting in the development of On-line Analytical Processing (OLAP). These two main categories shaped the database systems scenery in terms of what each system is addressing and what type of optimizations - and research in order to achieve them - are necessary for each category.

The crystallization of the goals of database systems had a very important implication. The design of database systems as software systems rapidly evolved to a standardized architecture (Hellerstein et al., 2007), which then allowed the researchers and the database systems developers to speak a common language. They were able to devise techniques that could be used in different systems because they were using the same high-level constructs despite having different implementations. This ease of interaction sped up the progress and the evolution of database systems and helped to build a large community and a large market with a variety of solutions optimized for different applications and use cases.

The architecture of a typical database system includes (1) the parser, (2) the optimizer, (3) the query engine, and (4) the storage manager. In a modern database system, the parser is responsible to parse a SQL query and analyze it syntactically, making sure it is consistent with the data at hand, and then, to forward it to the optimizer. The optimizer, in turn, evaluates what are the possible ways to execute the given query, and tries to find the fastest way (*optimal query plan*) to execute it.[2] After the query optimization phase, the query engine activates the chosen algorithms to execute the query and, when needed, it requests the data from the storage manager, which retrieves data from the memory and storage components.

The standardized architecture of database systems at the high-level enabled deep research for each and every module. Research on database systems covers query optimization, query execution, data access, but also transaction execution, security and distributed processing. The work presented in this thesis is related with query execution and techniques to access the data from the storage manager. A key observation is that the largest bulk of research on the internal modules of a database system is based on the assumption that the system uses hard disk drives as permanent storage. While there are approaches making different assumptions (e.g., in-memory databases with redundancy) the common denominator between most of the existing approaches is the presence of hard disks at some level to ensure durability of the data. Below we discuss how the trends in what data we generate and we analyze create new requirements for the internal modules of database systems motivating the research path taken.

1.3 Data Analytics

OLAP applications have been popular and crucial to business operation since the 1980s, however, the recent trends in the pace of data generation and the consequent needs for storing and analyzing data created a new set of requirements. Today, data have the following five key characteristics: (i) the data **volume** itself is large and growing exponentially, (ii) the data are constantly evolving and new information or updates are constantly generated, leading to high data **velocity**, (iii) a large **variety** of data sources include useful information for any given application making it very important to integrate diverse sources, (iv) as the data and the number of sources grow it is crucial to ensure the **veracity** of the data in order to have meaningful insights and (v) produce high **value** analysis which can help the scientific, business or organizational goals of the data analysis performed. Below we focus on the characteristics of volume, velocity, variety and value of data, to introduce data analytics requirements and the challenges that are addressed in this thesis.

---

[2] The optimizer is not always able to find the optimal query plan because it uses heuristics when searching for the appropriate combination of algorithms to minimize its response time.

1.3.1 Data Analytics Challenges

The above characteristics - volume, velocity, variety, veracity and value - are known as the *5 V's of Big Data*. "Knowledge is power", however, only when we know how to extract knowledge. Hence, the above characteristics combined with data analytics requirements create research challenges, which research on data analytics systems is attempting to address.

The sheer volume of the data and its velocity creates the *data freshness* requirement. The data management challenge is how to use the most recent version of the data when executing data analysis queries. Data freshness is a challenge because updating data and running analytical queries at the same time typically causes workload interference, mainly as a result of the disruption of analytical query access patterns (involving sequential scans of the hard disks) by scattered updates.

Velocity and value show that the value of our insights drops precipitously as the analysis queries are delayed. Hence, it is important to optimize for query response time and at the same time, transform the *destructive interference* to *collaborative execution* when possible.

Combining volume and velocity a new challenge is raised: how to *efficiently index*, and consequently access, continuously generated data - one very common use case. This challenge stems from the fact that traditional indexing techniques are built assuming no knowledge of the data attributes (e.g., distribution) since they mostly target data with no apriori knowledge or uniform distribution of the indexed values. Real-time data typically has a time dimension, on which it shows *implicit clustering* (Moerkotte, 1998), something that can be used to offer indexing with high performance and smaller size.

Variety surfaces the need to use and perform data analytics on data from diverse data sources including as much knowledge as possible for the question at hand. Knowledge-based datasets have been generated and are maintained in the context of the efforts of the semantic web. Typically, knowledge is represented using the Resource Description Framework (RDF) (RDf, 2013). Integrating the knowledge from RDF data in data analytics increases the value of the generated insights, however, the *different way to represent information in RDF* compared to the relational model makes it hard (i) to integrate RDF in the physical layer, and (ii) to exploit the research that has already been done on database system engines to execute RDF queries.

Before exploring the approaches taken to address these challenges we will dive in more details into which parts of the DBMS architectures play a key role in the aforementioned challenges.

1.4 Implications on the DBMS architecture

As we discussed in Section 1.2 the four layers of a DBMS are (i) the parser, (ii) the optimizer, (iii) the query engine, and (iv) the storage manager. Only the storage manager is copying data back and forth to its physical storage, typically hard disk drives (HDD). Nevertheless, the underlying assumption that data are ultimately stored on HDD plays a key role in the design of the optimizer and the query engine, in addition to the storage manager.

In particular, the query engine is equipped with algorithms implementing database operators - such as projection, selection, join, aggregates and so on - that are optimized for data residing on HDD. In turn, the optimizer selects the optimal variant of an algorithm using the cost assumptions of a HDD-equipped storage manager. The disk page assumption had to be maintained even when specialized algorithmic optimizations were proposed. For example, cache-conscious optimizations, like both the PAX data layout (Ailamaki et al., 2001) where a database disk page is internally organized in per-attribute minipages, and the Fractal B^+-Trees (Chen et al., 2002) where cache-optimized trees were embedded in disk-optimized trees, were optimized for cache performance, maintaining, however, the HDD page assumption. Moreover, even fundamental changes in the query engine architecture such as the design of pure column-store systems (Abadi, 2008; Boncz, 2002) are motivated by the characteristics of the memory hierarchy including both main memory and secondary storage. While HDD is the most commonly used technology for secondary storage, a wealth of new storage technologies has been under development (Freitas, 2009) the last few years creating new storage devices - Solid State Drives (SSD) - with different characteristics (more details in Section 2.1). By re-designing DBMS so as to exploit SSD we have the opportunity to augment their functionality and their capabilities to address the aforementioned data analytics challenges.

In fact, the data analytics challenges presented in Section 1.3.1 are connected very well with the storage layers used by DBMS. *Data freshness* hurts query response time because of the interference between the sequential reads of range scans and the random updates. Recent approaches that have been proposed to address this problem either increase the memory overhead of the system, or they cannot offer competitive performance (more details in Section 2.2.1). Supporting *analytical queries with high concurrency* can easily create contention of the resources of the storage layer. Tree indexing structures have been traditionally designed assuming that they will be stored on HDD, which have large and cheap capacity and slow random access performance. Finally, RDF management systems have been battling between an application-tailored storage solution and a DBMS backed-up storage solution. The first can offer typically better performance for a specific application, while the latter offers compatibility with other applications and uses the benefits of several decades of research on the storage layer and the query engine.

1.5 Solid-State Storage and Work Sharing for Efficient Scaleup Data Analytics

In this thesis we study the evolving requirements of data analytics engines and identify how new non-volatile solid state storage and query execution run-time optimizations help addressing these requirements. We outline the differences between traditional and solid-state storage and propose new algorithms, data organizations and indexing structures, that are solid-state aware. Furthermore, we study existing run-time optimizations based on work sharing, and we compare, integrate and redesign work sharing techniques.

To query and analyze data efficiently there have been two fundamental approaches: (i) distribution of the work in embarrassingly parallel tasks (when possible), and (ii) aggressive internal redesign of query engines. We extend these two approaches by adding in the equation the need of hardware-aware and, in particular, solid-state storage aware redesign of query engines focusing on optimizing the performance of a single node. The shared-nothing approach of finding embarrassingly parallel tasks was largely-based on the fact that in order to increase the processing power and the storage performance one would need the aggregate performance of multiple nodes. For example, parallelism was achieved by using several single-core machines and I/O bandwidth was achieved by huge arrays of disks or aggregate bandwidth of disks in different machines. The recent advancements of hardware, however, change these trends drastically. The evolution of solid-state storage with flash-based devices as the most popular example allows us to use a single solid-state device to achieve the same bandwidth as multiple disks and, at the same time, the trend towards multi-core and many-core processors gives huge processing power to a single node. Thus, we propose techniques to optimize the behavior of our systems in the single-node case - as an orthogonal direction of distributing the work to multiple nodes - by building hardware-aware, storage-aware and workload-aware query engines.

Using solid-state storage as an additional level in the memory hierarchy, combined with intelligent data sharing, we enable data analytics engines to improve throughput and response times, while maintaining data freshness.

1.5.1 Evolution of the storage hierarchy

The placement of solid-state storage in the memory hierarchy is an evolving research problem since the available technologies themselves are evolving. While the first widespread solid-state storage technology is flash, today a wealth of technologies is under research and development (Freitas, 2009), with Memristor and Phase Change Memory (PCM) being two of the most researched technologies. Flash-based SSD have been used either as a layer in between main memory and secondary storage or as alternative secondary storage. Future research of solid-state storage includes, however, studying the implication of using non-volatile main memory either exclusively or side-by-side with (volatile) main memory. Solid-state storage characteristics are different than HDD and main memory, nevertheless there exist partial commonalities. Therefore, integrating solid-state storage in the DBMS hierarchy requires designing new algorithms as well as reusing algorithms that were initially developed for other purposes.

The first step to understand how to integrate new storage technologies in a DBMS is to place these in the memory hierarchy. Traditionally, the top part of the memory hierarchy consists of the CPU registers and the cache memories, while moving towards its bottom part there is main memory, and various layers of secondary storage. The fundamental tradeoff between different levels of the memory hierarchy is capacity vs. access latency. The higher levels have small access latency while the lower levels have larger capacity. Contrary to the aforementioned memory hierarchy levels, solid-state storage equipped devices have asymmetric read and write latency. The asymmetry makes it unclear whether such devices should be yet another level of the hierarchy - in-between main memory and secondary storage - or they should be used side by side with main memory or secondary storage depending on the application.

In this thesis we synthesize the benefits of solid-state storage with the benefits of the other levels of the memory hierarchy. To that end, we use solid-state storage to exploit its fast random read performance and we complement the large capacity offered by hard disks. Moreover, we identify the need to redesign data structures and algorithms in order to take advantage the benefits of solid-state storage to offer to enhance data analytics performance, applicability and functionality.

We build *hardware-aware* data structures by studying the changes of the available storage technologies and by integrating efficiently to the existing systems. Furthermore, we build *workload-aware* algorithms which are capable of supporting the requirements of data analytics applications; data freshness and concurrency.

1.5.2 Summary and Contributions

Databases systems today are used for workloads with ever evolving requirements and characteristics. As more and more applications use the power of data analytics in their operations, data analytics systems face new challenges regarding the volume, the velocity, and the variety of data. We find that the new data analytics challenges can be addressed by careful integration of new solid-state storage technologies in the system.

This thesis makes the following key technical contributions:

1. **Efficient on-line updates in analytical queries.** We exploit fast random reads from SSDs and introduce the MaSM algorithms to support high-update rate with near-zero overhead in concurrent analytical queries. MaSM scales for different data sizes and delivers the same response time benefits for a variety of different storage configurations.
2. **Quantify the benefits of work sharing in data analytics.** We show that work sharing in the query engine of a data analytics system offers increased throughput and minimal latency overhead. As the number of concurrent queries in today's analytics applications increases, there are more opportunities to exploit work sharing to offer higher throughput with small latency overhead and sublinear increase in processing and storage resources.
3. **Competitive, space efficient approximate tree indexing.** We combine pre-existing workload knowledge of archival, social or monitoring data with probabilistic data structures to offer tailored indexing with smaller size footprint by one order of magnitude or more than traditional index structures, yet with competitive search performance.
4. **Data page layout for knowledge-based data.** We propose a new data layout for knowledge-

based data (using the RDF data model), which offers (i) increased locality for the properties of each subject, and (ii) low overhead for linkage queries. This approach outperforms the state-of-the-art for queries over a popular RDF datasets.

The work presented in this thesis, leading to the aforementioned technical contributions, serve as a platform to show the following key insights:

1. Solid-state storage helps us to offer increased functionality by complementing traditional hard disk drives with fast random reads and supporting reading sequential streams. The augmented functionality mitigates workload interference and changes the optimization goals since random reads are not the major bottleneck anymore.
2. Using solid-state storage devices as drop-in replacements of traditional storage does not lead to the desired benefits. On the contrary, successfully integrating solid-state storage devices in DBMS requires to (i) respect their limitations - e.g., avoid excessive number of writes and minimize random writes - and to (ii) use solid-state devices in tandem with hard disks using the strong points of both.
3. Sharing work amongst queries that have commonalities in their respective query plans is key to offering judicious resource utilization for workloads with increasing concurrency. We show that both (i) ad-hoc sharing, and (ii) batching queries using global query plans, offer stable performance with lower resource utilization than traditional query-centric execution.

This work paves the way for databases engines to address the requirements of emerging workloads by carefully integrating emerging storage technologies and redesigning execution algorithms. Furthermore, we believe that the methodology and the principles used generalize to other systems and applications that face the challenge to integrate new storage hardware in their stack.

1.5.3 Published papers

The material in this thesis has been the basis for a number of publications in major international refereed venues in the area of databases and database management systems.

1. M. Athanassoulis, A. Ailamaki, S. Chen, P. Gibbons and R. Stoica. Flash in a DBMS: Where and How?, in **IEEE Data Engineering Bulletin**, vol. 33, num. 4, 2010.
2. M. Athanassoulis, S. Chen, A. Ailamaki, P. Gibbons and R. Stoica. MaSM: Efficient Online Updates in Data Warehouses. ACM **SIGMOD** International Conference on Management of Data, Athens, Greece, 2011.
3. M. Athanassoulis, B. Bhattacharjee, M. Canim and K. A. Ross. Path Processing using Solid State Storage. 3rd International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (**ADMS**), Istanbul, Turkey, 2012.

4. I. Psaroudakis, M. Athanassoulis and A. Ailamaki. Sharing Data and Work Across Concurrent Analytical Queries. 39th International Conference on Very Large Data Bases (**VLDB**), 2013.

5. M. Athanassoulis and A. Ailamaki. BF-Tree: Approximate Tree Indexing. Submitted for publication.

6. M. Athanassoulis, S. Chen, A. Ailamaki, P. Gibbons and R. Stoica. Online Updates on Data Warehouses via Judicious Use of Solid-State Storage. Submitted for publication.

1.6 Outline (How to read this thesis)

The rest of the thesis is organized as follows. Chapter 2 discusses previous research on data analytics engines, storing and managing RDF data, and work sharing. Moreover, Chapter 2 details key hardware and storage trends that drive design decision in database systems. Regarding data freshness in analytical queries, we discuss relevant work which inspired the proposed algorithms, and its shortcomings when it comes to offering the necessary design guarantees for solid-state storage. In Chapter 2 we further discuss indexing techniques for archival data and indexing techniques for data stored on solid-state storage and we outline how the proposed approximate indexing combines the best of both words. Finally, Chapter 2 discusses the state-of-the-art for storing and managing RDF data and for implementing work-sharing in query engines.

After reading Chapter 2, Chapters 3-6 can be read independently. Chapter 3 shows how to exploit the fast random reads of solid-state disks in order to support high-update rate and concurrent analytical queries with near-zero overhead. We introduce MaSM, a set of algorithms to organize data on SSDs and merge them efficiently combining SSDs random read performance, and HDD sequential scan performance with an efficient external merging scheme. We use a solid-state storage cache to store the incoming updates, which are organized in sorted runs, similar to the sort order of the main data. Any query that needs to use some of the cached updates merges at run-time the updates with main data. We model this merge as an external merge-sort join. We show that the proposed techniques achieve negligible overhead in the query execution time and increase the sustained update rate by orders of magnitude when compared with in-place updates.

Chapter 4 discusses how existing work-sharing techniques for analytical query engines can be used to use the available resources efficiently in the presence of analytical workloads with high number of concurrent queries. We compare the two main categories of work-sharing, we integrate them and propose a different approach of ad-hoc work-sharing which make work-sharing always beneficial. We find that existing approaches for ad-hoc sharing are based on the assumption of uni-core processors, hence failing to deliver performance benefits in the multi-core case.

Chapter 5 shows how the combination of workload knowledge and probabilistic data structures enables us to build a space efficient, yet competitive tree index tailored for solid-state storage. The proposed index, BF-Tree, departs from both paradigms of indexes for archival data and solid-state aware indexes and addresses the union of the requirements of the two. The proposed tree structure exploits the implicit clustering or ordering of archival data and stores location information with high granularity, using a probabilistic data structure (Bloom filters). This results in a small increase in number of unnecessary reads which can be tolerated either if the underlying storage does not suffer from random access interference (i.e., if it is solid-state storage), or if the false positives are very low and on average do not hurt performance. We show that on average the proposed tree structure can achieve competitive search times, with smaller index size, when compared with traditional indexing techniques.

Chapter 6 introduces a new data page organization, RDF-tuple, in order to allow knowledge-based data, stored as RDF data, to be compatible with traditional data analytics systems. RDF-tuple offers better performance by increasing locality and minimizing unnecessary reads when traversing the data. We build a prototype repository which used the proposed data layout. We show that it offers competitive performance in a set of experiments against the research state-of-the-art system and, further, investigate the impact of the proposed data layout when used on top of novel storage such as flash-based and PCM-based [3] solid-state drives.

Chapter 7 discusses some of the future research goals regarding integrating solid-state storage and building run-time optimizations in data analytics engines. We discuss the forthcoming changes in terms of the storage hierarchy and their impact in data analysis systems design. We outline research directions based on deeper integration of solid-state storage in the DBMS stack. Moreover, we observe the similar pattern of challenges that computer systems will face when persistent main memory becomes available, and we outline the parallel between DBMS and persistent storage, and operating systems and persistent main memory. Then, we discuss how work sharing techniques can be used in the context of column-stores or hybrid (between row and column oriented storage) data analysis systems. Last but not least, we discuss the benefits of integrating solid-state storage in the storage hierarchy used in the context of systems employing work sharing.

---

[3] See Chapter 2 for more details on the storage technologies under development.

2 Related Work and Background

In this thesis we highlight the importance of understanding the characteristics and the requirements of the workloads before we decide how to best exploit solid-state storage. There has been a lot of related work regarding the usage of flash as caching layers and regarding specialized optimizations in accessing data (e.g. flash-aware indexing structures). Our study works towards the thesis that the variability of storage technologies – and hence of the behavior of storage devices – leads to the necessity to assess how solid-state storage can address the requirements of each workload and we present four studies for workloads with different characteristics.

2.1 Hardware Trends in Storage

2.1.1 Fifty years of hard disks

For the past fifty years, hard disk drives (HDDs) have been the building blocks of storage systems. They have been the slowest component of many computer systems, because of long data access times as well as limitations to the maximum bandwidth offered. HDDs have the slowest increase in performance compared to main memory and CPU, and the gap widens with every hardware generation. Since the 1980s, HDD's access time is almost the same and bandwidth lags behind capacity growth (in 1980, reading an entire disk took about a minute; the same operation takes four hours today as shown in Table 2.1). CPU power increased dramatically by orders of magnitude, memory size and memory bandwidth grew exponentially, and larger and deeper cache hierarchies are increasingly successful in hiding main memory latency. The reasons behind this trend lie deeply in the physical limitations of HDDs where moving parts put a hard constraint on how fast data can be accessed and retrieved (HDD's performance is dictated by the *seek time* due to the movement of the arm and the *rotational delay* due to the rotation of the platter (Ramakrishnan and Gehrke, 2002, Chapter 9.1.1)).

Figure 2.1 shows in more detail the evolution of hard disks over the last fifty years. In the figure we can see the evolution of five metrics throughout time, from 1956 where IBM introduced the

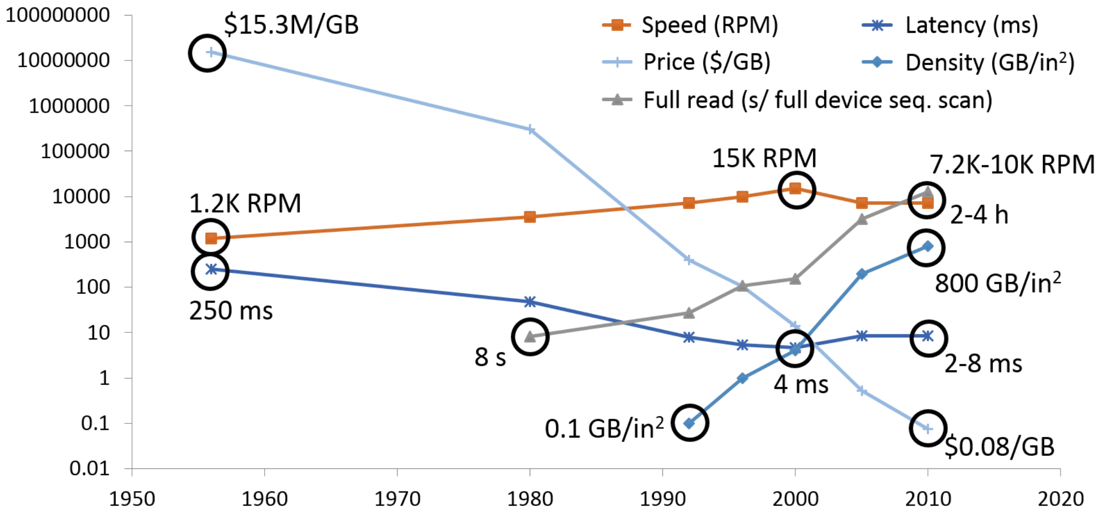


Figure 2.1: Fifty years of HDD

first hard disk until the devices that are available today in the 2010s. Note that the x-axis of the figure is time in terms of years and the y-axis is a logarithmic axis for every metric. The first line (light blue) corresponds to the price per storage capacity which we quantify as US dollars per gigabyte. Extrapolating the price of the 1956 IBM disk, which could hold about 3.5MB, the price per GB was 15.3M USD/GB. Today, the price has dropped by eight orders of magnitude to 0.08$/GB. The second line (orange) shows the trend in terms of the rotational speed. We observe that there has been a slow but steady increase from 1.2K rotations per minute (RPM) in 1956 to 15K RPM in 2000. Since then, however, the rotational speed stagnated due to practical reasons (friction and power consumption) and today's disks typically have between 7.2K RPM and 10K RPM, mainly because having more has too high power consumption. The third line (dark blue) shows how latency evolved from 250ms for the 1956 device to 2-8ms for the latest devices. Latency is a function of rotational speed, hence, since 2000 there has been no significant improvement, other than building devices with smaller form factor which require smaller movements. The fourth line (gray) shows the time needed to perform a full sequential scan over a device with the representative size of its time. Essentially, *this line shows that hard disks performance, contrary to their capacity, stopped scaling*. While in the 1980s an entire hard disk needed only eight second to be read, today we would require anywhere between two and four hours. The fifth and final line (medium darkness blue) shows the evolution of the density of the devices for the last two decades that there is accurate information. This trend corroborates that device capacity still scales: from 1990s until today the density has increased about four orders of magnitude from 0.1GB/in^2 to 800GB/in^2.

2.1.2 From hard disks to solid-state storage

Today it is clear that HDD technology does not provide the improvement needed to pace with increasing compute power, and rotating drives will be eventually replaced by other technologies.

The best candidate for this task, flash memory, has been getting significant momentum over the past few years and is becoming the de facto storage medium for increasingly more applications. Flash memory started as a storage solution for small consumer devices two decades ago and has evolved into high-end storage for performance sensitive enterprise applications. Flash promises to fill in the performance gap created by HDD and also has several side benefits such as lower power consumption and better mechanical reliability. A comparison of several characteristics of a state-of-the-art flash device of the 2010s with traditional hard disk devices from the 1980s and the 2010s is presented in Table 2.1. Flash is perhaps the best known HDD challenger as a pervasive storage technology but there are also other competing technologies like phase-change memory (PCM) developed by IBM, memristor developed by HP, and others yet immature technologies backed by the biggest hardware manufactures worldwide (Burr et al., 2008).

Device	Capacity (MB)	*Seq. BW (MB/s)*	Rnd acc. (ms)	*Rnd acc. (MB/s)*	**Ratio**	Seq. Scan (s)	Cost ($/MB)	Cost ($)
HDD 1980	100	*1.2*	28.3	*0.28*	**1:4.35**	83	200	20000
HDD 2010	1000000	*80*	8	*0.98*	**1:81.92**	12500	0.0003	300
Flash 2010	100000	*700*	0.026	*300*	**1:2.33**	143	0.02	2000

Table 2.1: HDD trends compared and flash characteristics

As solid-state devices (SSD) become increasingly adopted by several applications – including data analytics (Barroso, 2010) – the sheer amount of data stored on SSD increases exponentially thus, creating the need to create appropriate techniques to store, manage, retrieve and process data from SSD.

2.1.3 Flash-based solid-state storage

Since the introduction of flash memory in the early 90's, flash storage devices have gradually spread to more and more applications and are now a standard storage solution in embedded devices, laptops and even desktop PCs. This trend is driven by the exponential growth of flash chip capacity (flash chips follow Moore's law as they are manufactured using the same techniques and equipment as integrated circuits), which drives down exponentially their price per GB.

Flash devices have fundamental differences compared with hard disks. First, flash devices have no moving parts and random accesses are not penalized by long seek times or rotational delays, so sequential read accesses are no longer necessary. Second, asymmetric performance between reads and writes arises because NAND memory cells cannot be written directly (updated in place) but instead require slow and costly erase-then-write operations. In addition, NAND cell erasure must be performed for a large number of flash pages (typical values today are 64-256 pages of 512B-4KiB each), while pages can be read individually. Whenever a page is written its physical neighbors must be moved elsewhere to allow the erase operations, causing additional data copies and exacerbating the performance asymmetry between reads and writes. Third, because of the elevated voltage required by an erase operation, NAND cells can only tolerate a limited number of erasures. To prolong the life of flash chips, the firmware must spread erase operations evenly to avoid wearing out individual areas and use error correction codes (ECCs) to ensure data integrity.

Flash device manufacturers try to hide the aforementioned issues behind a dedicated controller embedded in the flash drive. This abstraction layer, called the Flash Translation Layer (FTL), deals with the technological differences and presents a generic block device to the host OS. The main advantage of an FTL is that it helps to maintain backward compatibility, under the assumption that the generic block device is the prevailing abstraction at the application level. In practice, however, most performance-critical applications developed over the last 30 years are heavily optimized around the model of a rotating disk. As a result, simply replacing a magnetic disk with a flash device does not yield optimal device or DBMS performance, and query optimizer decisions based on HDD behavior become irrelevant for flash devices. Moreover, the complex and history-dependent nature of the interposed layer affects predictability.

2.1.4 Phase Change Memory

PCM stores information using resistance in different states of phase change materials: amorphous and crystalline. The resistance in the amorphous state is about five orders of magnitude higher than the crystalline state, and it differentiates between 0 (high resistance) and 1 (low resistance) (Chen et al., 2011; Papandreou et al., 2011). Storing information on PCM is performed through two operations: set and reset. During the set operation, current is applied on the device for a sufficiently long period to crystallize the material. During the reset operation higher current is applied for shorter duration in order to melt the material and then cool it abruptly, leaving the material in the amorphous state. Unlike flash, PCM does not need the time consuming erase operation to write the new value. PCM devices can employ a simpler driver than the complex FTL that flash devices use to address the wear leveling and performance issues (Akel et al., 2011). In the recent literature there is already a discussion about how to place PCM in the existing memory hierarchy. While proposed ideas (Chen et al., 2011) include placing PCM side-by-side DRAM as an alternative non-volatile main memory, or even using PCM as the main memory of the system, current prototype approaches consider PCM as a secondary storage device providing PCIe connectivity. There are three main reasons why this happens: (i) the endurance of each PCM cell is typically 10^6–10^8, which is higher than flash (10^4–10^5 with a decreasing trend (Abraham, 2010)) but still not enough for a main memory device, (ii) the only available interface to date is PCIe, and (iii) the PCM technology is new, so the processors and the memory hierarchy do not yet have the appropriate interfaces for it.

The Moneta system is a hardware-implemented PCIe storage device with PCM emulation by DRAM (Caulfield et al., 2010, 2012). Performance studies on this emulation platform have highlighted the need for improved software I/O latency in the operating system and file system. The Onyx system (Akel et al., 2011) replaces the DRAM chips of Moneta with first-generation PCM chips,[1] yielding a total capacity of 10 GB. Onyx is capable of performing a 4KB random read in $38\mu s$ and a 4KB write request[2] in $179\mu s$. For a hash-table based workload, Onyx

---

[1] The PCM chips used by Onyx are the same as those used in our profiled device, but the devices themselves are different.

[2] The write numbers for Onyx use early completion, in which completion is signaled when the internal buffers have

performed 21% better than an ioDrive, while the ioDrive performed 48% better than Onyx for a B-Tree based workload (Akel et al., 2011).

The software latency (as a result of operating system and file system overheads) is measured to be about 17μs (Akel et al., 2011). On the other hand, the hardware latency for fetching a 4K page from a hard disk is on the order of milliseconds and for a high-end flash device is about 50μs. Early PCM prototypes need as little as 20μs to read a 4K page increasing the software contribution in relative latency from 25% (17μs out of 67μs) for a flash device like FusionIO, to 46% (17μs out of 37μs) for a PCM prototype. Minimizing the impact of software latency is a relatively new research problem acknowledged by the community (Akel et al., 2011). Caulfield et al. (Caulfield et al., 2012) point out this problem and propose a storage hardware and software architecture to mitigate the overheads to take better advantage of low latency devices such as PCM. The architecture provides a private, virtualized interface for each process and moves file system protection checks into hardware. As a result, applications can access file data without operating system intervention, eliminating OS and file system costs entirely for most accesses. The experiments show that the new interface improves latency and bandwidth for 4K writes by 60% and 7.2x respectively, OLTP database transaction throughput by up to 2.0x, and Berkeley-DB throughput by up to 5.7x (Caulfield et al., 2012).

Jung et al. (Jung et al., 2011) ran the `fio` profiler over the same Micron PCM prototype available to us and a popular flash device (OCZ Revodrive), showing qualitative differences between the PCM device and the flash device. The PCM device, unlike the flash device, shows no difference between latency for random and sequential accesses for different values of IO depth (number of concurrent outstanding requests) or page sizes.

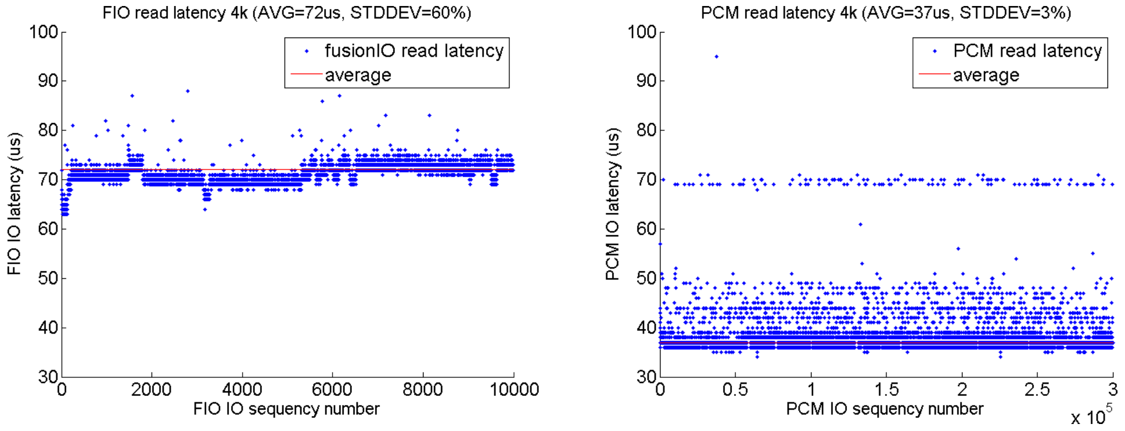
Lee et al. (Lee et al., 2010) introduce a new In-Page Logging (IPL) design that uses PCRAM as a storage device for log records. They claim that the low latency and byte addressability of PCRAM can allow one to avoid the limitations of flash-only IPL. Papandreou et al. (Papandreou et al., 2011) present various programming schemes for multilevel storage in PCM. The proposed schemes are based on iterative write-and-verify algorithms that exploit the unique programming characteristics of PCM in order to achieve significant improvements in resistance-level packing density, robustness to cell variability, programming latency, energy per-bit and cell storage capacity. They present experimental results from PCM test-arrays to validate the proposed programming schemes. In addition, the reliability issues of multilevel PCM in terms of resistance drift and read noise are discussed.

2.1.5 Flash vs. PCM

In this section we perform a head-to-head comparison between a state-of-the-art PCI-based FusionIO flash device and an early PCI-based PCM-based prototype. This comparison provides

---

enough information to complete the write, but before the data is physically in the PCM. Early completion is also used by SSD devices, supported by large capacitors to ensure that the writes actually happen in case of a power failure.



(a) (b)

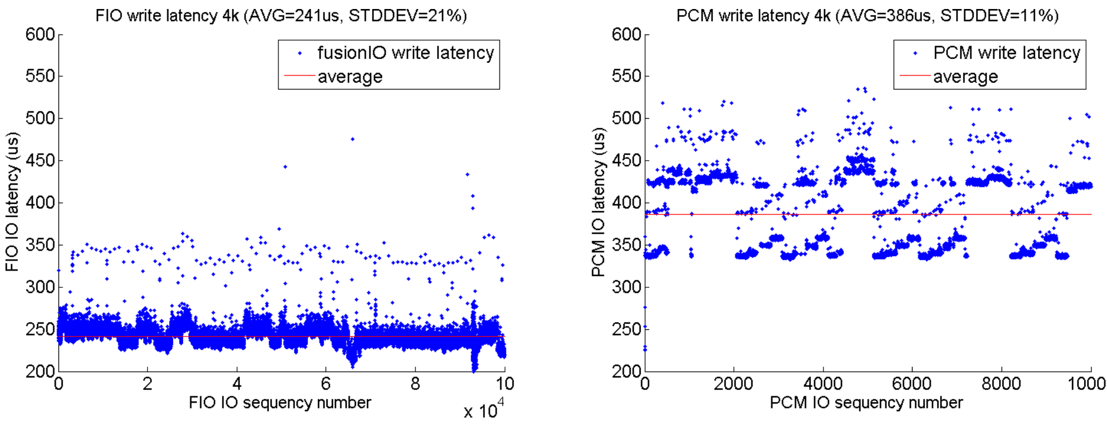
Figure 2.2: Read latency for (a) flash (zoomed in the most relevant part) and (b) PCM

some early insights about what PCM-based devices can offer when compared with existing flash-based devices and as a roadmap of what we can expect from future PCM-based devices.

Experimental methodology. The experimental setup is the same as described in Section 6.2. We show that PCM technology has already an important advantage for read-only workloads. We present several experiments comparing the read latency of the aforementioned devices. In particular, we measure the latency for direct access to the devices bypassing any caching (operating system or file system) using the O_DIRECT and O_SYNC flags in a custom C implementation.

Read latency. The first experiment measures the read latency per 4K I/O request using flash and PCM. We perform ten thousand random reads directly to the device. In the flash case, there are a few outliers with orders of magnitude higher latency, a behavior encountered in related literature as well (Bouganim et al., 2009; Chen, 2009). Figure 2.2(a) is in fact a zoomed in version of the overall data points (excluding outliers) in order to show the most interesting part of the graph; there is some variation between 65μs and 90μs. The average read latency for 4K I/Os using flash is 72μs. The standard deviation of the read latency in flash is 60% of the average read latency. The PCM device, however, behaves differently both qualitatively and quantitatively. Firstly, there are no outliers with orders of magnitude higher read latency. The values are distributed between 34μs and 95μs with the vast majority (99%) focused in the area 34–40μs (Figure 2.2(b)), averaging 36μs. Secondly, the standard deviation in terms of a percentage of the average is much smaller compared with flash, only 3%. The first two experiments show clearly that the very first PCM devices will already be competitive for latency-bound read-intensive workloads both because of the average performance and the increased stability and predictability of performance.

Write latency. For write intensive workloads the available PCM prototype performs worse than the popular flash representative device, but it is still more stable. In particular, the PCM device



(a) (b)

Figure 2.3: Write latency for (a) flash (zoomed in the most relevant part) and (b) PCM

has average write latency 386μs with 11% standard deviation while the flash device has average write latency 241μs with 21% standard deviation, which is depicted in Figures 2.3(a) and (b). On the other hand, the basic PCM firmware can support only 40MB/s writes. This prototype limitation in write performance is attributed to the fact that it was designed with read performance in mind. Newer PCM chips, however, are expected to increase write speed significantly over time, consistent with the raw capability of the technology, bridging the gap in write performance at the device level. Note that the flash device uses a complex driver (the result of 3 or more years of optimizations and market feedback) while the PCM prototype uses a naive driver that provides basic functionality.

The measured read and write latency numbers are consistent with numbers from Onyx (Akel et al., 2011), except that write latency is higher because the PCM device at hand does not use early completion.

Discussion. PCM is becoming extensively prototyped and tested as a candidate for either main memory or flash replacement. The characteristics of PCM are expected to match main memory in terms of access latency and density, however, the initial applications for PCM have been targeting techniques to replace flash. PCM, like flash, is non volatile and future research questions include *how to exploit non volatile main memory*.

2.1.6 More solid-state storage technologies

While flash and PCM are two of the most researched and developed solid-state storage technologies, there is a plethora of other technologies that are being pursued. A few examples include variations of flash (SONOS and TANOS flash), ferroelectric RAM, magnetic RAM, resistive RAM and more (Burr et al., 2008; Freitas, 2009). Another solid-state storage technology under development is Memristor which is based on circuits that have variable resistance (Strukov et al., 2008).

2.1.7 Sequential Reads and Random Writes in Storage Systems

Concurrently with this thesis, Schindler et al. (Schindler et al., 2011) proposed exploiting flash as a non-volatile write cache in storage systems for efficient servicing of I/O patterns that mix sequential reads and random writes. Compared to the online update problem in data warehouses, the settings in storage systems are significantly simplified: (i) an “update record” in storage systems is (a new version of) an entire I/O page and (ii) ACID is not supported for accessing multiple pages. As a result, the proposal employs a simple update management scheme: modifying the I/O mapping table to point to the latest version of pages. Interestingly, the proposal exploits a disk access pattern, called *Proximal I/O*, for migrating updates that write to only 1% of all disk pages.

2.2 Data Analysis Systems

Data analytics provides business and research insight from a massive collection of data being continuously augmented. Historically, however, updates to the data were performed offline using bulk insert/update features – executed mainly during extensive idle-times (e.g., at night). The quick reacting, 24x7 nature of business today cannot tolerate long delays in updates, because the value of the insights gained drops fast when the data used are stale (Inmon et al., 2003). Thus, data analysis systems supporting efficient real-time updates have been the goal of recent research and business efforts (Becla and Lim, 2008; Oracle, 2004, 2013; White, 2002; Polyzotis et al., 2008). State-of-the-art data analysis systems, however, today often fail to support fast analysis queries over fresh data.

A demanding data-intensive application is the utilization of systematically organized knowledge to increase the quality of the performed analysis. The semantic web (Shadbolt et al., 2006) is an effort to allow storing, managing and searching machine readable information using the Resource Description Framework (RDF) (RDf, 2013) model to represent the data. Several applications, including scientific (Apweiler et al., 2004), business and governmental (Data.gov, 2013; Data.gov.uk, 2013), use this data representation model, a trend which is strengthened by efforts like Linked Open Data (LOD) (Bizer et al., 2009a). LOD, to date, consists of more than 25 billion RDF triples collecting data from more than 200 data sources. Data analytics engines recently started considering semantic RDF databases as sources for their analysis. A celebrated example of the combination of traditional data analytics, knowledge representation and reasoning and other techniques (e.g., machine learning) is the IBM Watson Computer (Ferrucci et al., 2010) which participated successfully in the knowledge game "Jeopardy!". Querying RDF datasets, however, poses a new challenge for data analytics engines: efficiently querying semantic data that form graphs which have edges for both semantic and relational connections.

The majority of data stored and used by data analytics engines are stored based on a specific dimension, often a time one. Accessing ordered, or partitioned, data efficiently by data analytics engines is already a design goal by the Netezza (Francisco, 2011) engine. Moreover, the advent of solid-state storage results in storing more and more data on such devices. The capacity, however, is much more expensive in terms of $ per GB compared with that of traditional hard disks. Hence, a key problem is indexing data with small capacity requirements taking into account that this data are often sorted or partitioned.

2.2.1 High-Throughput Updates and Data Analysis

The need to augment the functionality of analysis systems with high-throughput update capability has been recognized in the literature.

In-Memory and External Data Structures. In Chapter 3 we present MaSM, a novel approach to offer online updates in data analytics with the usage of solid-state storage. MaSM extends prior work on in-memory differential updates (Héman et al., 2010; Stonebraker et al., 2005) to overcome the limitation on high migration costs vs. large memory footprint. We assume I/O to be the main bottleneck for data warehousing queries and therefore the focus of our design is mainly on the I/O behaviors. On the other hand, prior differential update approaches propose efficient in-memory data structures, which is orthogonal to the MaSM design, and may be applied to MaSM to improve CPU performance for CPU-intensive workloads.

Handling Updates with Stacked Tree Structures. The state-of-the-art approach to support online updates with minimal overhead is differential updates. Positional delta trees (Héman et al., 2010) provide efficient maintenance and ease in merging differential updates in column stores. They defer touching the read-optimized image of the data as long as possible and buffer differential updates in a separate write-store. The updates from the write-store are merged with the main data during the column scans in order to present to the user an up-to-date version of the data. O'Neil et al. (1996) proposed log-structured merge tree (LSM) which is organized as a collection of trees (two or more) stacked vertically, supporting high update rate of indexing structures of a database, like B-trees. The first level of LSM is always in-memory and the last on disk. The updates are initially saved in the first level and they are consequently moved to the next levels (in batches) in order to reach the last level. Searching using LSM requires to traverse all stacked trees as each one is not inclusive of the subsequent trees. While LSM is capable of sustaining a high update rate, and – if tuned appropriately – delivering good read performance, it applies more writes per update than MaSM as we detail in Chapter 3 (Section 3.4.5).

LSM has inspired other approaches that maintain, index and query data. FD-Tree (Li et al., 2010) is a tree-structure which extends LSM by reducing the read overhead and by allowing flash-friendly writes. The key ideas are to limit random writes to a small part of the index and to transform random writes to sequential writes. FD-Tree assumes a fixed ratio between two consecutive levels in order to offer the best insertion cost amortization, which is the optimal case for LSM as well. As we discuss in Chapter 3 (Sections 3.2.3 and 3.4.5), this setup incurs an increased number of writes per update since every update is propagated a number of times.

The stepped-merge algorithm (Jagadish et al., 1997) stores updates lazily in a B$^+$-Tree organization by first maintaining updates in memory in sorted runs and, eventually, form a B$^+$-Tree of these updates using an external merge-sort. The stepped-merge approach aims at minimizing random I/O requests. On the other hand, MaSM focuses on minimizing main memory consumption and the unnecessary writes on the flash device at the expense of more, yet efficient, random read I/O requests.

The quadtree-based storage algorithm (McCarthy and He, 2011) was proposed concurrently and independently of the MaSM algorithms presented in Chapter 3. It uses in-memory Δ-quadtrees to organize incoming updates for a data cube. The Δ-quadtrees are used to support the computation of specific operations like SUM, COUNT and AVG over a predefined data cube. When the available memory is exhausted, the Δ-quadtrees are flushed to an SSD. Subsequent range queries are answered by merging the Δ-quadtrees in memory, the Δ-quadtrees on flash, and the main data in the data cube. Contrary to the quadtree-based storage algorithm, MaSM is not part of the aggregate calculation allowing for a general purpose solution to handle incoming updates in a DW.

Extraction-Transformation-Loading for Data Warehouses. We focus on supporting efficient query processing given online, well-formed updates. An orthogonal problem is an efficient ETL (Extraction-Transformation-Loading) process for data warehouses (Oracle, 2013; Polyzotis et al., 2008). ETL is often performed at a data integration engine outside the data warehouse to incorporate changes from front-end operational data sources. Streamlining the ETL process has been both a research topic (Polyzotis et al., 2008; Simitsis et al., 2009) and a focus of a data warehouse product (Oracle, 2013). These ETL solutions can be employed to generate the well-formed updates to be applied to the data warehouse.

2.2.2 Adaptive organization of data and updates

Database cracking and adaptive indexing. There is a wealth of differential approaches to maintain indexes and other data structures in order to enhance query performance. Database cracking (Idreos et al., 2007) reorganizes the data in memory to match how queries access data. Adaptive indexing (Graefe and Kuno, 2010; Graefe et al., 2012) follows the same principle as database cracking by focusing the index optimization on key ranges used in actual queries. Their hybrids for column stores (Idreos et al., 2011) take the best of the two worlds. Contrary to MaSM, database cracking, adaptive index, and their hybrids focus on static data which are dynamically reorganized (or the corresponding indexes optimized) according to the incoming queries. The MaSM algorithms enable long running queries to see no performance penalty by allowing concurrent updates.

Partitioned B-trees. Graefe proposed partitioned B-trees (Graefe, 2003) in order to offer efficient resource allocation during index creation, or data loading into an already indexed database. The core idea is to maintain an artificial leading key column to a B-tree index. If each value of this column has cardinality greater than one, the index entries are effectively partitioned. Such an indexing structure (i) permits storing all runs of an external sort together, (ii) reduces substantially the wait time until a newly created index is available and (iii) solves the dilemma whether one should drop the existing index or update the existing index one-record-at-a-time when large amount of data is added to an already indexed large data warehouse. Partitioned B-trees can work in parallel with MaSM in order to offer efficient updating of the main data as well as efficient indexing in data warehouses.

2.2.3 Orthogonal Uses of SSDs for Data Warehouses

Orthogonal to the research presented in this thesis, previous studies have investigated placing objects (such as data and indexes) on SSDs vs. disks (Canim et al., 2009; Koltsidas and Viglas, 2008; Ozmen et al., 2010), including SSDs as a caching layer between main memory and disks (Canim et al., 2010b), and as temporary storage for enhancing performance of non-blocking joins (Chen et al., 2010).

2.3 Optimizing Data Analysis Queries Using Work-Sharing

In this section, we provide the necessary background on work sharing techniques in the literature. We start by shortly reviewing related work, and continue to extensively review work related to Simultaneous Pipelining (SP) and Global Query Plans (GQP), which compose our main area of interest. In Table 2.2, we summarize the sharing methodologies used by traditional query-centric systems and the research prototypes we examine.

System	Traditional query-centric model	QPipe	CJOIN	DataPath	SharedDB
Sharing in the execution engine	Query Caching, Materialized Views, MQO	Simultaneous Pipelining	Global Query Plan (joins of Star Queries)	Global Query Plan	Global Query Plan (with Batched Execution)
Sharing in the I/O layer	Buffer pool management techniques	Circular scan of each table	Circular scan of the fact table	Asynchronous linear scan of each disk	Circular scan of in-memory table partitions
Storage Manager	Any	Any	Any	Special I/O Subsystem (read-only requests)	Crescando (read and update requests)

Table 2.2: Sharing methodologies employed by query-centric models and the research prototypes we examine.

2.3.1 Sharing in the I/O layer and in the execution engine

Sharing in the I/O layer. By sharing data, we refer to techniques that coordinate and share the accesses of queries in the I/O layer. The typical query-centric database management system (DBMS) incorporates a buffer pool and employs eviction policies (Chou and DeWitt, 1985; Johnson and Shasha, 1994; Megiddo and Modha, 2003; O'Neil et al., 1993). Queries, however, communicate with the buffer pool manager on a per-page basis, thus it is difficult to analyze their access patterns. Additionally, if multiple queries start scanning the same table at different times, scanned pages may not be re-used.

For this reason, shared scans have been proposed. Circular scans (Colby et al., 1998; Cook, 2001; Morri, 2002; Harizopoulos et al., 2005) are a form of shared scans. They can handle a large number of concurrent scan-heavy analytical queries as they reduce contention for the buffer pool and the number of I/O requests to the underlying storage devices. Furthermore, more elaborate shared scans can be developed for servicing different fragments of the same table or different groups of queries depending on their speed (Lang et al., 2007; Zukowski et al., 2007), and for main-memory shared scans (Qiao et al., 2008).

Shared scans can be used to handle a large number of concurrent updates as well. The Crescando (Unterbrunner et al., 2009) storage manager uses a circular scan of in-memory table partitions, interleaving the reads and updates of a batch of queries along the way. The scan first executes the update requests of the batch for a scanned tuple in their arrival order, and then the read requests.

Shared scans, however, are not immediately translated to a fast linear scan of a disk. The DataPath system (Arumugam et al., 2010) uses an array of disks and stores relations column-by-column by randomly hashing pages to the disks. During execution, it reads pages from the disks asynchronously and in order, thus aggregating the throughput of the linear scan of each disk.

Sharing in the execution engine. By sharing work among queries, we refer to techniques that avoid redundant computations inside the execution engine. A traditional query-centric DBMS typically uses query caching (Shim et al., 1999) and materialized views (Roussopoulos, 1982). Both, however, do not exploit sharing opportunities among in-progress queries.

Multi-Query Optimization (MQO) techniques (Roy et al., 2000; Sellis, 1988) are an important step towards more sophisticated sharing methodologies. MQO detects and re-uses common sub-expressions among queries. The disadvantages of classic MQO are that it operates on batches of queries only during optimization, and that it depends on costly materializations of the common intermediate results. This cost can be alleviated by using pipelining (Dalvi et al., 2003), which additionally exploits the parallelization provided by multicore processors. The query plan is divided into sub-plans and operators are evaluated in parallel.

Both SP and GQP leverage forms of pipelined execution and sharing methodologies which bear some superficial similarities with MQO. These techniques, however, provide deeper and more dynamic forms of sharing, at run-time, among in-progress queries. In the rest of this section, we

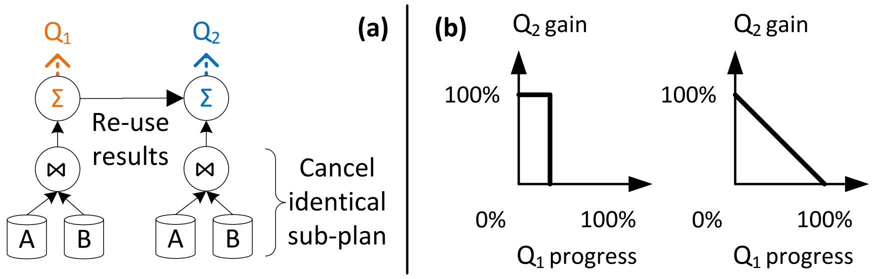


Figure 2.4: (a) SP example with two queries having a common sub-plan below the join operator. (b) A step and a linear WoP.

provide an extended overview of both SP and GQP, and the systems that introduce them: QPipe (Harizopoulos et al., 2005) and CJOIN (Candea et al., 2009, 2011) respectively. We also mention how more recent research prototypes advanced the GQP technique.

2.3.2 Simultaneous Pipelining

SP identifies identical sub-plans among concurrent queries at run-time, evaluates only one and pipelines the results to the rest simultaneously (Harizopoulos et al., 2005). We depict in Figure 2.4a an example of two queries that share a common sub-plan below the join operator (along with any selection and join predicates), but have a different aggregation operator above the join. SP evaluates only one of them, and pipelines the results to the other aggregation operator.

Fully sharing common sub-plans is possible if the queries arrive at the same time. Else, sharing opportunities may be restricted. The amount of results that a newly submitted $Q2$ can re-use from the ***pivot operator*** (the top operator of the common sub-plan) of the in-progress $Q1$, depends on the type of the pivot operator and the arrival of $Q2$ during $Q1$'s execution. This relation is expressed as a ***Window of Opportunity*** (WoP) for each relational operator (following the original acronym (Harizopoulos et al., 2005)). In Figure 2.4b, we depict two common WoP, a step and a linear WoP.

A step WoP expresses that $Q2$ can re-use the full results of $Q1$ if it arrives before the first output tuple of the pivot operator. Joins and aggregations have a step WoP. A linear WoP signifies that $Q2$ can re-use the results of $Q1$ from the moment it arrives up until the pivot operator finishes. Then, $Q2$ needs to re-issue the operation in order to compute the results that it missed before it arrived. Sorts and table scans have a linear WoP. In fact, the linear WoP of the table scan operator is translated into a circular scan of each table.

2.3.3 The QPipe execution engine

QPipe (Harizopoulos et al., 2005) is a relational execution engine that supports SP at execution time. QPipe is based on the paradigms of staged databases (Harizopoulos and Ailamaki, 2003).

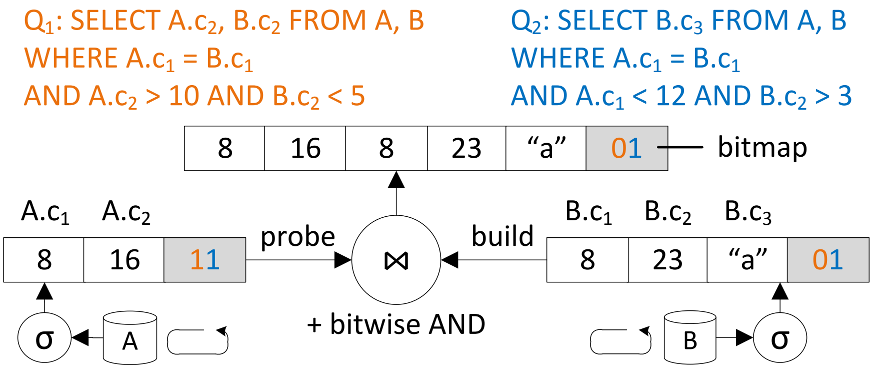


Figure 2.5: Example of shared selection and hash-join operators.

Each relational operator is encapsulated into a self-contained module called a ***stage***. Each stage has a queue for work requests and employs a local thread pool for processing the requests.

An incoming query execution plan is converted to a series of inter-dependent ***packets***. Each packet is dispatched to the relevant stage for evaluation. Data flow between packets is implemented through FIFO (first-in, first-out) buffers and page-based exchange, following a push-only model with pipelined execution. The buffers also regulate differently-paced actors: a parent packet may need to wait for incoming pages of a child and, conversely, a child packet may wait for a parent packet to consume its pages.

This design allows each stage to monitor only its packets for detecting sharing opportunities efficiently. If it finds an identical packet, and their inter-arrival delay is inside the WoP of the pivot operator, it attaches the new packet (***satellite*** packet) to it (***host*** packet). While it evaluates the host packet, SP copies the results of the host packet to the output FIFO buffer of the satellite packet.

2.3.4 Global Query Plans with shared operators

SP is limited to common sub-plans. If two queries have similar sub-plans but with different selection predicates for the involved tables, SP is not able to share them. Nevertheless, the two queries still share a similar plan that exposes sharing opportunities. It is possible to employ ***shared operators***, where a single shared operator can evaluate both queries simultaneously. The basic technique for enabling them is sharing tuples among queries and correlating each tuple to the queries, e.g. by annotating tuples with a ***bitmap***, whose bits signify if the tuple is relevant to one of the queries.

The simplest shared operator is a shared selection, that can evaluate multiple queries that select tuples from the same relation. For each received tuple, it toggles the bits of its attached bitmap according to the selection predicates of the queries. A hash-join can also be easily shared by queries that share the same equi-join predicate. In Figure 2.5, we show a conceptual example of how a single shared hash-join is able to evaluate two queries. It starts with the build phase by receiving tuples from the shared selection operator of the inner relation. Then, the probe phase begins by receiving tuples from the shared selection operator of the outer relation. The hash-join proceeds as normal, by additionally performing a bit-wise AND between the bitmaps of the joined tuples.

The most significant advantage is that a single shared operator can evaluate many more similar queries. For example, a shared hash-join can evaluate many queries having the same equi-join predicate, and possibly different selection predicates. In the worst case, the union of the selection predicates may force it to join the whole two relations. The disadvantage of a shared operator in comparison to a query-centric one is that it reduces parallelism in order to gain throughput, entailing increased book-keeping. For example, a shared hash-join maintains a hash table for the union of the tuples of the inner relation selected by all queries, and performs bit-wise operations between the bitmaps of the joined tuples. For low concurrency, as shown by our experiments (see Section 4.5), query-centric operators outperform shared operators. A similar tradeoff is found for the specific case of shared aggregations on CMP (Cieslewicz and Ross, 2007).

By using shared scans and shared operators, a GQP can be built for evaluating all concurrent queries. GQP are introduced by CJOIN (Candea et al., 2009, 2011), an operator based on shared selections and shared hash-joins for evaluating the joins of star queries (Kimball and Ross, 2002). GQP are advanced by the DataPath system (Arumugam et al., 2010) for more general schemas, by tackling the issues of routing and optimizing the GQP for a newly incoming query. DataPath also adds support for a shared aggregate operator, that calculates a running sum for each group and query.

Both CJOIN and DataPath handle new queries immediately when they arrive. This is feasible due to the nature of the supported shared operators: selections, hash-joins and aggregates. Some operators, however, cannot be easily shared. For example, a sort operator cannot easily handle new queries that select more tuples than the ones being sorted (Arumugam et al., 2010). To overcome this limitation, SharedDB (Giannikis et al., 2012) batches queries for every shared operator. Batching allows standard algorithms to be easily extended to support shared operators, as they work on a fixed set of tuples and queries. SharedDB supports shared sorts and various shared join algorithms, not being restricted only to equi-joins. Nevertheless, batched execution has drawbacks: A new query may suffer increased latency, and the latency of a batch is dominated by the longest-running query.

2.3.5 The CJOIN operator

The selection of CJOIN (Candea et al., 2009, 2011) for our analysis is based on the facts that it introduced GQP, and that it is optimized for the simple case of star queries. Without loss of generality, we restrict our evaluation to star schemas, and correlate our observations to more general schemas (used, e.g., by DataPath (Arumugam et al., 2010) or SharedDB (Giannikis et al.,

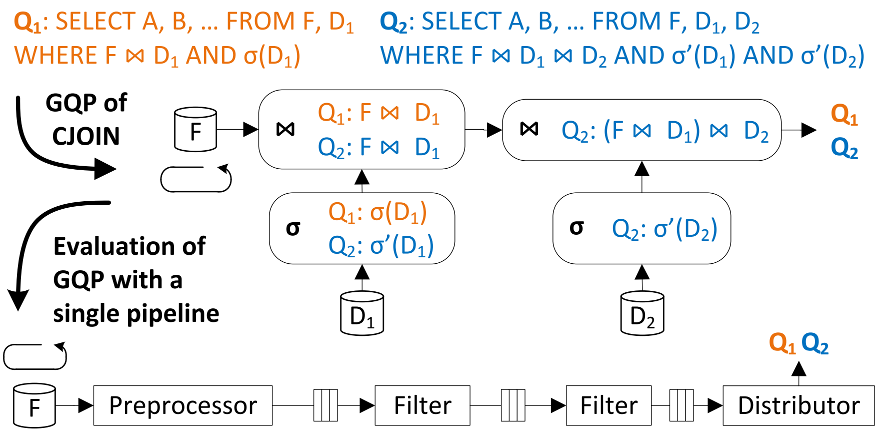


Figure 2.6: CJOIN evaluates a GQP for star queries.

2012)).

Star schemas are very common for organizing data in relational DW. They allow for numerous performance enhancements (Kimball and Ross, 2002). A star schema consists of a large ***fact table***, that stores the measured information, and is linked through foreign-key constraints to smaller ***dimension tables***. A ***star query*** is an analytical query over a star schema. It typically joins the fact table with several dimension tables and performs operations such as aggregations or sorts.

CJOIN evaluates the joins of all concurrent star queries, using a GQP with shared scans, shared selections and shared hash-joins. In Figure 2.6, we show the GQP that CJOIN evaluates for two star queries. CJOIN adapts the GQP with every new star query. If a new star query references already existing dimension tables, the existing GQP can evaluate it. If a new star query joins the fact table with a new dimension table, the GQP is extended with a new shared selection and hash-join. Due to the semantics of star schemas, the directed acyclic graph of the GQP takes the form of a chain.

CJOIN exploits this form to facilitate the evaluation of the GQP. It materializes the small dimension tables and stores in-memory the selected dimension tuples in the hash tables of the corresponding shared hash-joins. Practically, for each dimension table, it groups the shared scan, selection and hash-join operators into an entity called ***filter***. When a new star query is admitted, CJOIN pauses, adds newly referenced filters, updates already existing filters, augments the bitmaps of dimension tuples according to the selection predicates of the new star query, and then continues.

Consequently, CJOIN is able to evaluate the GQP using a single pipeline: the ***preprocessor*** uses a circular scan of the fact table, and flows fact tuples through the pipeline. The data flow in the pipeline is regulated by intermediate FIFO buffers, similar to QPipe. The filters in-between are actually the shared hash-joins that join the fact tuples with the corresponding dimension tuples and additionally perform a bit-wise AND between their bitmaps. At the end of the pipeline, the

distributor examines the bitmaps of the joined tuples and forwards them to the relevant queries. For every new query, the preprocessor admits it, marking its point of entry on the circular scan of the fact table and signifies its completion when it wraps around to its point of entry on the circular scan.

2.3.6 Quantifying work sharing opportunities

Earlier in this section we introduced the fundamental approaches in sharing in relational database systems, i.e., in the storage layer and in the query engine. The motivation stems from the fact that concurrent queries access common base data and perform common database operations on the data as pointed out by the MQO work (Sellis, 1988). In addition, as concurrency in analytical workloads increases exponentially (Russom, 2012) the opportunities of sharing increase as well. A fundamental question, however, is to quantify the opportunities of work sharing or the headroom of improvement through work sharing techniques. Using the definition of work sharing by Johnson et al. (2007) as *"any operation that reduces the total amount of work in a system by eliminating redundant computation or data accesses"* the need to define a metric to quantify the opportunity for performance or resource utilization improvement surfaces. In fact, a principled framework for work sharing opportunities can also be the basis for a fundamentally new multi-query optimization. In the literature to date one can find detailed experiments and measurements on the performance benefits of specific work sharing implementations in terms of response time, throughput or even resource utilization. Such analysis can be found for the DataPath system (Arumugam et al., 2010), the CJOIN operator (Candea et al., 2011), the QPipe query engine (Harizopoulos et al., 2005; Johnson et al., 2007), the integration of CJOIN into QPipe (Psaroudakis et al., 2013) and the SharedDB query engine (Giannikis et al., 2012). A first step to work sharing headroom quantification is the MQO and its different approaches (Sellis, 1988; Chen and Dunham, 1998), which was further advanced by a practical realization (Roy et al., 2000). Existing work, however, does not include an analysis for what would be the maximum possible benefit given a workload and up to what extent does each specific approach realize this benefit.

In addition, in the context of stream processing work sharing has been extensively studied (Chen et al., 2000; Madden et al., 2002; Chandrasekaran et al., 2003; Agrawal et al., 2008; Liu et al., 2008; Majumder et al., 2008). Precision sharing (Krishnamurthy et al., 2004) introduces a very important relevant aspect: work sharing without introducing unnecessary work, which is key in order to quantify correctly the work sharing opportunities. Similarly to work sharing literature for database engines, existing work for work sharing in stream processing, does not include an analysis to quantify the work sharing opportunities.

2.4 Bloom filters

A Bloom filter (BF) (Bloom, 1970) is a space-efficient probabilistic data structure supporting membership tests with non-zero probability for false positives (and zero probability for false negatives). Typically in a BF one can only add new elements and never remove elements. A deletable BF has been discussed (Rothenberg et al., 2010) but it is not generally adopted.

2.4.1 Bloom filters' applications in data management

BFs have been extensively used in networking (Broder and Mitzenmacher, 2002; Lim and Kim, 2010) and as auxiliary data structures in database systems (Antognini, 2008; Chang et al., 2006; Condie et al., 2010; Dean and Ghemawat, 2004; Mullin, 1990). Modern database systems utilize BFs while implementing several algorithms, like semi-joins (Mullin, 1990), where BFs help in implementing faster the join algorithm. Google's Bigtable (Chang et al., 2006) uses BFs to reduce the number of accesses to internal storage components and a study (Antognini, 2008) shows that Oracle systems use BFs for tuple pruning, to reduce data communication between slave processes in parallel joins and to support result caches.

2.4.2 Bloom filters for evolving workloads and storage

BFs have been used for changing workloads and different storage technologies. Scalable Bloom Filters (Almeida et al., 2007) study how a BF can adapt dynamically to the numbers of elements stored while assuring a bound for maximum false positive probability. Buffered Bloom filters (Canim et al., 2010a) use flash, as well, as a cheaper alternative to main memory. The BF is allocated in blocks equal to the typical block size of the storage device and both reads and writes are buffered in memory to improve locality during the build and the probe phase. The forest-structured Bloom filter (Lu et al., 2011) aims at designing an efficient flash-based BF by using the memory to capture the frequent updates. Memory is used to capture the updates and hide them from the flash storage and, on flash, the BF is organized in a forest-structure in order to improve the lookup performance. Bender et al. (Bender et al., 2012) propose three variations of BFs: the quotient filter, the buffered quotient filter and the cascade filter. The first uses more space than traditional BF but shows better insert/lookup performance and supports deletes. The last two variations are designed on top of quotient filter supporting larger workloads, serving, as well, as alternatives of BF for SSD.

In Chapter 5 we extend the usage of BFs in DBMS by proposing BF-Tree, an indexing tree structure which uses BFs to trade off capacity for indexing accuracy by applying approximate indexing. BF-Tree is orthogonal to the optimizations described above. In fact, a BF-Tree can take advantage of such optimizations in order to fine-tune its performance.

2.5 Indexing for Data Analysis

Both analytical workloads and storage technologies evolve, and as a result an increasing amount of data is stored on SSDs. In Chapter 5 we propose an SSD-aware indexing for analytical workloads which departs from the traditional SSD-aware index structure. We do so by creating a new tree index structures which is tailored for SSD by understanding that SSD is the diametrical opposite than HDD in two aspects: random access performance and capacity price. The proposed structure combines capacity and performance optimizations of different prior approaches.

2.5.1 SSD-aware indexing

We find that approximate indexing is suitable for modern storage devices (e.g. flash or PCM-based) because of their principal difference compared to traditional disks: random accesses perform virtually the same as sequential accesses[3]. Since the rise of flash as an important competitor of disks for non-volatile storage (Gray, 2007) there have been several efforts in creating a flash-friendly indexing structure (often a flash-aware version of a B^+-Tree). LA-Tree (Agrawal et al., 2009) uses lazy updates, adaptive buffering and memory optimizations to minimize the overhead of updating flash. In μ-Tree (Kang et al., 2007) the nodes along the path from the root to the leaf are stored in a single flash memory page in order to minimize the number of flash write operations during the update of a leaf node. IPLB^+-tree (Na et al., 2011) avoids costly erase operations - often caused by small random write requests common in database applications - in order to improve the overall write performance. SILT (Lim et al., 2011) is a flash-based memory-efficient key-value store based on cuckoo hashing and tries, which offers fast search performance using minimal amount of main memory.

What SSD-aware indexing does not do. Related work focuses, mostly, on optimizing for specific flash characteristics (read/write asymmetry, lifetime) maintaining the same high-level indexing structure and does not address the shifting trade-off in terms of capacity. The BF-trees we present in this thesis, orthogonally to flash optimizations, trade off capacity for indexing accuracy.

2.5.2 Specialized optimizations for tree indexing

SB-Tree (O'Neil, 1992) is designed to support high-performance sequential disk access for long range retrievals. It is designed assuming an array of disks from which it retrieves data or intermediate nodes should they not be in memory, by employing multi-page reads during sequential access to any node level below the root. Moreover, SB-Tree and B^+-Tree have similar capacity requirements. On the contrary, BF-Tree aims at indexing partitioned data, at decreasing the size of the index, and at taking advantage of the characteristics of SSD. Litwin and

---

[3] In early flash devices random reads were one order of magnitude faster than random writes (Bouganim et al., 2009; Stoica et al., 2009), but later devices are more balanced.

Lomet (Litwin and Lomet, 1986) introduce a generic framework for index methods called the bounded disorder access method. The method, similarly to BF-Trees, aims at increasing the ratio of file size to index size (i.e., to decrease the size of the index) by using hashing for distributing the keys in a multi-bucket node. The bounded disorder access method does not decrease the index size aggressively for good reason, since by doing that it would cause more random accesses on the storage medium which is assumed to be traditional hard disks. On the contrary, BF-Trees propose aggressive reduction of the index size by utilizing a probabilistic membership-testing data structure as a form of compression. The performance of such an approach is competitive because the SSD can sustain several concurrent read requests with zero or small penalty (Athanassoulis et al., 2012; Roh et al., 2011) and the penalty of (randomly) reading pages due to a false positive is lower on SSD compared with HDD.

2.5.3 Indexing for data warehousing

Efficiently indexing of ordered data having small space requirements is recognized as a desired feature by commercial systems. Typically data warehousing systems use some form of sparse indexes to fulfill this requirement. The Netezza data warehousing appliance uses ZoneMap acceleration (Francisco, 2011), a lightweight method to maintain a coarse-index, to avoid scanning rows that are irrelevant to the analytic workload, by exploiting the natural ordering of rows in a data warehouse. Netezza's ZoneMap indexing structure is based on the Small Materialized Aggregates (SMA) earlier proposed by Moerkotte (Moerkotte, 1998). SMA is a generalized version of a Projection Index (O'Neil and Quass, 1997). A Projection Index is an auxiliary data structure containing an entire column - similar to how column-store systems (MonetDB, 2013; Vertica, 2013) today physically store data. Instead of saving the entire column, SMA store an aggregate per group of contiguous rows or pages, thus, enhancing queries calculating aggregates than can be inferred by the stored aggregate. In addition to primary indexes or indexes on the order attribute, secondary, hardware-conscious data warehouse indexes have been designed to limit access to slow IO devices (Sidirourgos and Kersten, 2013).

2.6 Knowledge-based Data Analysis

In this section we present related work in terms of representing knowledge-based data using the Resource Description Framework (RDF) data model, as well as relevant benchmarks and the systems that are typically used to store, manage and query RDF data.

2.6.1 RDF datasets and benchmarks

The wide adoption of RDF format has led to the design of numerous benchmarks and datasets (Duan et al., 2011) each one focusing on the different usage scenarios of RDF data. Benchmarks include:

- LUBM (Guo et al., 2005): a benchmark consisting of university data.
- BSBM (Bizer and Schultz, 2009): a benchmark built around an e-commerce data use case that models the search and navigation pattern of a consumer looking for a product.
- SP2Bench (Schmidt et al., 2009): a benchmark based on the DBLP database of article publications, modelling several search patterns.
- Yago2 (Hoffart et al., 2011, 2013): data from Wikipedia, WordNet and GeoNames.
- UniProt (Apweiler et al., 2004): a comprehensive, high-quality and freely accessible database comprised of protein sequences.
- DBpedia (Bizer et al., 2009b): a dataset consisting of data extracted from Wikipedia and structured in order to make them easily accessible.

Several of the aforementioned benchmarks and workloads include path processing queries that could be inefficiently evaluated if the graph-like form of data is not taken into account. Viewing RDF data as relational data may make it more difficult to apply optimizations for graph-like data access patterns such as search. While each triple can conceptually be represented as a row, it has more information than a single row since it signifies a relation between two nodes of a graph. Two neighboring nodes may end up in the same search very often (a simple locality case) or nodes connected with two neighboring nodes may end up in the same search often. Moreover, a specific path between two nodes can be the core of the query. In this case, and especially if there are few or no branches in the path, evaluating a path of length k as $k-1$ joins can substantially increase the cost of answering one such query.

2.6.2 Storing RDF data

RDF data are comprised by statements, each represented as a triple of the form <*Subject, Predicate, Object*>. Each triple forms a statement which represents information about the $Subject$. In particular, $Subject$ is connected to the $Object$ using a specific $Predicate$ modelling either a connection[4] between the $Subject$ and the $Object$ or the value of a property of the $Subject$.[5] In fact, in RDF triples, sometimes, the $Predicate$ is called $Property$ and the $Object$ is called $Value$. We will maintain the terminology <*Subject, Predicate, Object*>.

RDF data form naturally sparse graphs but the underlying storage model in existing systems is not always tailored for graph processing (Hassanzadeh et al., 2011). There are two trends as far as how to physically store RDF data: (i) using as underlying storage a relational database system (either a row-store (Alexander et al., 2005; Neumann and Weikum, 2008) or a column

---

[4] For example, the triple <*Alice, isFriendWith, Bob*> shows that a friendship connection between $Alice$ and Bob exists.

[5] For example, the triple <*Alice, birthday, 01/04/1980*> shows that the property $birthday$ of $Alice$ has value $01/04/1980$.

store (Abadi et al., 2007)) or (ii) design a native store (Virtuoso, 2011; Wilkinson, 2006), a storage system designed particularly for RDF data which can be tailored to the needs of a specific workload. Support for RDF storage and processing assuming an underlying relational data management system is proposed from the industrial perspective (Bhattacharjee et al., 2011) as well.

The RDF data layout we present in this thesis is different in three ways. First, it does not assume a traditional relational data layout but only the notion of variable sized tuples (having in effect a variable number of columns). Second, while our approach resembles prior art (Bhattacharjee et al., 2011) as far as storing several triples with the same $Subject$ (or $Object$) physically close by, it is not bound by the limitations of relational storage, and it avoids repetition of information (e.g., for the $Objects$ that are connected to a specific $Subject$ with the same $Property$ the identifier of the $Property$ is stored only once). Third, we depart from the relational execution model, which is vital because graph traversals using relational storage lead to repetitive self-joins. We can support optimized graph-traversal algorithms without paying the overheads that come with relational query evaluation.

2.7 Summary

In this chapter we briefly discussed very interesting research and technological advancements that promoted solid-state storage to a key player in the memory hierarchy research, which in turn is tightly connected with the design and the characteristics of data management and data analytics systems.

In the path to efficiently integrate solid-state storage in data analytics systems we present our contributions in a storage-aware, workload-aware system. In the next Chapters we present in detail our approaches towards ensuring data freshness, supporting concurrency and new archival workloads, and integrating diverse data sources.

In the last chapter, we discuss the role of storage in data management and data analytics systems. We attempt to lay the groundwork of future research directions that will become necessary to pursue in order to exploit what the future technologies offer to meet what the future applications need.

3 Online Updates for Data Analytics using Solid-State Storage[1]

3.1 Introduction

Data warehouses (DW) are typically designed for efficient processing of *read-only* analysis queries over large data. Historically, updates to the data were performed using bulk insert/update features that executed offline—mainly during extensive idle-times (e.g., at night). Two important trends lead to a need for a tighter interleaving of analysis queries and updates. First, the globalization of business enterprises means that analysis queries are executed round-the-clock, eliminating any idle-time window that could be dedicated to updates. Second, the rise of online and other quickly-reacting businesses means that it is no longer acceptable to delay updates for hours, as older systems did: the business value of the answer often drops precipitously as the underlying data becomes more out of date (Inmon et al., 2003; White, 2002). In response to these trends, data warehouses must now support a much tighter interleaving of analysis queries and updates, so that analysis queries can occur 24/7 and take into account very recent data updates (Becla and Lim, 2008). The large influx of data is recognized as one of the key characteristics of modern workloads, often referred to as *Velocity* of data (Zikopoulos et al., 2012). Thus, Active (or Real-Time) Data Warehousing has emerged as both a research topic (Polyzotis et al., 2008) and a business objective (Oracle, 2013; White, 2002; IBM, 2013; Russom, 2012) aiming to meet the increasing demands of applications for the latest version of data. Unfortunately, state-of-the-art data warehouse management systems *fall short of the business goal of fast analysis queries over fresh data*. A key unsolved problem is how to efficiently execute analysis queries in the presence of *online* updates that are needed to preserve data freshness.

---

[1] The material of this chapter has been the basis for the IEEE Data Engineering Bulletin paper "Flash in a DBMS : Where and How ?" (Athanassoulis et al., 2010), the ACM SIGMOD 2011 paper "MaSM : Efficient Online Updates in Data Warehouses" (Athanassoulis et al., 2011), and for a paper submitted for publication entitled "Online Updates on Data Warehouses via Judicious Use of Solid-State Storage" (Athanassoulis et al., 2013).

3.1.1 Efficient Online Updates for DW: Limitations of Prior Approaches

While updates can proceed concurrently with analysis queries using concurrency control schemes such as snapshot isolation (Berenson et al., 1995), the main limiting factor is the physical interference between concurrent queries and updates. We consider the two known approaches for supporting *online updates*[2], in-place updates and differential updates, and discuss their limitations.

In-Place Updates Double Query Time. A traditional approach, used in OLTP systems, is to update in place, i.e., to store the new value in the same physical location as the previous one. However, as shown in Section 3.2.2, in-place updates can dramatically slow down data warehousing queries. Mixing random in-place updates with TPC-H queries increases the execution time, on average, by 2.2x on a commercial row-store data warehouse and by 2.6x on a commercial column-store data warehouse. In the worst case, the execution time is 4x longer. Besides having to service a second workload (i.e., the updates), the I/O sub-system suffers from the *interference* between the two workloads: the disk-friendly sequential scan patterns of the queries are disrupted by the online random updates. This factor alone accounts for 1.6x slowdown on average in the row-store DW.

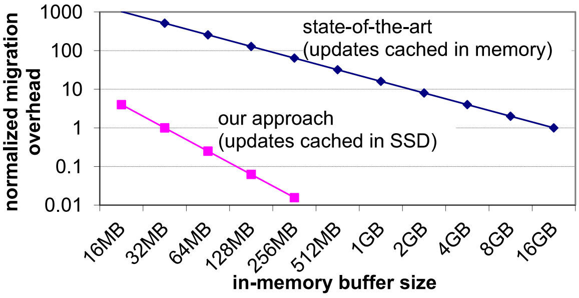


Figure 3.1: An analysis of migration overheads for differential updates as a function of the memory buffer size. Overhead is normalized to the prior state-of-the-art using 16GB memory.

Differential Updates Limited by In-Memory Buffer. Recently, *differential updates* have been proposed as a means to enable efficient online updates in column-store data warehouse (Héman et al., 2010; Stonebraker et al., 2005), following the principle of differential files (Severance and Lohman, 1976). The basic idea is to (i) cache incoming updates in an in-memory buffer, (ii) take the cached updates into account on-the-fly during query processing, so that queries see fresh data, and (iii) migrate the cached updates to the main data whenever the buffer is full. While these proposals significantly improve query and update performance, their reliance on an *in-memory* buffer for the cached updates poses a fundamental trade-off between migration overhead and memory footprint, as illustrated by the “state-of-the-art” curve in Figure 3.1 (note: log-log scale, the lower the better). In order to halve the migration costs, one must double the in-memory buffer size so that migrations occur (roughly) half as frequently. The updates that are typically distributed to the entire file, hence, a large buffer would cache updates corresponding to virtually

---

[2] We focus on online updates for real-time analytics, hence, we do not consider solutions designed for offline updates such as master and transactional tapes.

the entire data warehouse. Each migration is expensive, incurring the cost of scanning the entire data warehouse, applying the updates and writing back the results (Stonebraker et al., 2005; Héman et al., 2010). However, dedicating a significant fraction of the system memory solely to buffering updates degrades query operator performance as less memory is available for caching frequently accessed data structures (e.g., indexes) and storing intermediate results (e.g., in sorting, hash-joins). Moreover, in case of a crash, the large buffer of updates in memory is lost, prolonging crash recovery.

Caching updates on HDD. Apart from caching updates in-memory, one can use HDD to cache updates. Fundamentally, DW updates can be maintained on HDD using several approaches based on differentials file and log-structure organizations (Severance and Lohman, 1976; Jagadish et al., 1997; O'Neil et al., 1996; Graefe, 2003, 2006b; Graefe and Kuno, 2010). Any approach to store updates on HDD will suffer from the performance limitations of HDDs. A key factor is the performance of the disk caching the updates compared with the performance of the disk storing the main data. If main data and cached updates reside on the same physical device, we will observe workload interference similar to what happens for the in-place updates in Section 3.2.2. If updates are cached on a separate HDD, its capabilities cannot support both *high update rate* and *good query response times*, since the incoming updates will interfere with the updates to be read for answering the queries. A third design is to use a separate high-end storage solution, such as an expensive disk array, for caching updates. Such a storage setup is able to offer very good read and write performance, but is also quite expensive. The introduction of flash-based Solid State Drive (SSD) presents a new design point. SSD is much less expensive than HDD based storage for achieving the same level of random read performance. In the following, we exploit SSDs for caching updates, aiming to achieve online updates with negligible overhead on query performance. In Section 3.1.2 we outline our approach.

3.1.2 Our Solution: Cache Updates in SSDs for online updates in DWs

We exploit the recent trend towards including a small amount of (SSDs) in mainly HDD-based computer systems (Barroso, 2010). Our approach follows the idea of differential updates discussed above, but instead of being limited to an in-memory buffer, makes judicious use of SSDs to cache incoming updates. Figure 3.2 presents the high-level framework. Updates are stored in an SSD-based update cache, which is 1%–10% of the main data size. When a query reads data, the relevant updates on SSDs are located, read, and merged with the bulk of the data coming from disks. A small in-memory buffer is used as a staging area for the efficient processing of queries and incoming updates. The updates are migrated to disks only when the system load is low or when updates reach a certain threshold (e.g., 90%) of the SSD size.

Design Goals. We aim to achieve the following five design goals:

1. *Low query overhead with small memory footprint:* This addresses the main limitations of

prior approaches.

2. *No random SSD writes:* While SSDs have excellent sequential read/write and random read performance, random writes perform poorly because they often incur expensive erase and wear-leveling operations (Bouganim et al., 2009). Moreover, frequent random writes can transition an SSD into sub-optimal states where even the well-supported operations suffer from degraded performance (Bouganim et al., 2009; Stoica et al., 2009).

3. *Low total SSD writes per update:* A NAND flash cell can endure only a limited number of writes (e.g., 10^5 writes for enterprise SSDs). Therefore, the SSDs' lifetime is maximized if we minimize the amount of SSD writes per incoming update.

4. *Efficient migration:* Migrations should occur infrequently while supporting high sustained update rate. A naive approach is to migrate updates to a new copy of the data warehouse and swap it in after migration completes, essentially doubling the disk capacity requirement. We want to remove such requirement by migrating to the main data in place.

5. *Correct ACID support:* We must guarantee that traditional concurrency control and crash recovery techniques still work.

Prior differential update approaches (Stonebraker et al., 2005; Héman et al., 2010) maintain indexes on the cached updates in memory, which we call Indexed Updates (IU). We find that naïvely extending IU to SSDs incurs up to 3.8x query slowdowns (Section 3.2.3 and 3.5.2). While employing log structured merge-trees (LSM) (O'Neil et al., 1996) can address many of IU's performance problems, LSM incurs a large number of writes per update, significantly reducing the SSDs' lifetime (Section 3.2.3).

At a high level, our framework is similar to the way many popular key-value store implementations (e.g., Bigtable (Chang et al., 2006), HBase (HBase, 2013), and Cassandra (Lakshman and Malik, 2010)) handle incoming updates by caching them in HDDs and merging related updates into query responses. In fact, their design follows the principle of LSM. However, the design of these key-value stores is focused on neither low overhead for data warehousing queries with small memory footprint, using SSDs and minimizing SSD writes, nor correct ACID support for multi-row transactions. Using SSDs instead of HDDs for the update cache is crucial to our design, as it reduces range scan query overhead by orders of magnitude for small ranges.

Using SSDs to support online updates: MaSM. We use MaSM (*materialized sort-merge*) algorithms that achieve the five design goals with the following techniques.

First, we observe that the "Merge" component in Figure 3.2 is essentially an outer join between the main data on disks and the updates cached on SSDs. Among various join algorithms, we find that sort-merge joins fit the current context well: cached updates are sorted according to the layout order of the main data and then merged with the main data. MaSM exploits external sorting algorithms both to achieve *small memory footprint* and to *avoid random writes* to SSDs.

To sort $\|SSD\|$ pages of cached updates on SSD, two-pass external sorting requires $M = \sqrt{\|SSD\|}$ pages of memory. The number of passes is limited to two because one of the design goals is to minimize the amount of writes our algorithms perform on the SSDs. Compared with differential updates limited to an in-memory update cache, the MaSM approach can effectively use a small memory footprint, and exploits the larger on-SSD cache to greatly reduce migration frequency, as shown in the "our approach" curve in Figure 3.1.

Second, we optimize the two passes of the "Merge" operation: generating sorted runs and merging sorted runs. For the former, because a query should see all the updates that an earlier query has seen, we materialize and reuse sorted runs, amortizing run generation costs across many queries. For the latter, we build simple read-only indexes on materialized runs in order to reduce the SSD read I/Os for a query. Combined with the excellent sequential/random read performance of SSDs, this technique successfully achieves *low query overhead* (at most only 7% slowdowns in our experimental evaluation).

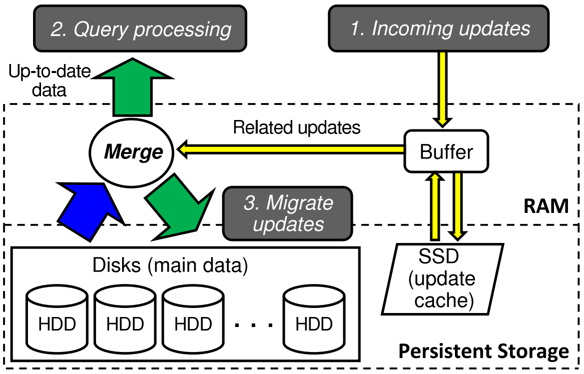


Figure 3.2: Framework for SSD-based differential updates.

Third, we consider the *trade-off between memory footprint and SSD writes*. The problem is complicated because allocated memory is used for processing both incoming updates and queries. We first present a MaSM-2M algorithm, which achieves the minimal SSD writes per update, but allocates M memory for incoming updates and M memory for query processing. Then, we present a more sophisticated MaSM-M algorithm that reduces memory footprint to M but incurs extra SSD writes. We select optimal algorithm parameters to minimize SSD writes for MaSM-M. After that, we generalize the two algorithms into a MaSM-αM algorithm. By varying α, we can obtain a spectrum of algorithms with different trade-offs between memory footprint and SSD writes.

Fourth, in order to support *in-place migration* and *ACID properties*, we propose to attach timestamps to updates, data pages, and queries. That is, there is a timestamp per update when it is cached on SSDs, a timestamp per data page in the main data, and a timestamp for every query. Using timestamps, MaSM can determine whether or not a particular update has been applied to a data page, thereby enabling concurrent queries during in-place migration. Moreover, MaSM guarantees serializability in the timestamp order among individual queries and updates. This can be easily extended to support two-phase locking and snapshot isolation for general transactions involving both queries and updates. Furthermore, crash recovery can use the timestamps to determine and recover only the updates in the memory buffer (updates on SSDs survive the crash).

Finally, we minimize the impact of MaSM on the DBMS code in order to reduce the development effort to adopt the solution. Specifically, MaSM can be implemented in the storage manager (with minor changes to the transaction manager if general transactions are to be supported). It does not require modification to the buffer manager, query processor or query optimizer.

3.1.3 Contributions

This chapter makes the following contributions:

- First, to our knowledge, this is the first effort to *exploits SSDs for efficient online updates in data warehouses*. We propose a high-level framework and identify five design goals for a good SSD-based solution.
- Second, we propose the family of MaSM algorithms that exploits a set of techniques to successfully achieve the five design goals.
- Third, we study the trade-off between memory footprint and SSD writes with MaSM-2M, MaSM-M, and MaSM-αM.
- Fourth, we study the trade-off between memory footprint and short range query performance for all MaSM algorithms. We provide a detailed model to predict MaSM query performance for a variety of different system setups.
- Fifth, we present an experimental study of MaSM's performance.

Our results show that MaSM incurs only up to 7% overhead both on range scans over synthetic data (varying range size from 4KB to 100GB) and in a TPC-H query replay study, while also increasing the sustained update throughput by orders of magnitude. Finally, we discuss various DW-related aspects, including shared nothing architectures, Extract-Transform-Load (ETL) processes, secondary indexes, and materialized views.

Outline. Section 3.2 sets the stage for our study. Section 3.3 presents the proposed MaSM design. Section 3.4 presents the proofs of the performance guarantees and analytical modelling of the proposed techniques. Section 3.5 presents the experimental evaluation. Section 3.6 discusses DW-related issues. Section 3.7 concludes the chapter.

3.2 Efficient Online Updates and Range Scans in Data Warehouses

In this section, we first describe the basic concepts and clarify the focus of our study. As in most optimization problems, we would like to achieve good performance for the frequent use cases, while providing correct functional support in general. After that, we analyze limitations of prior approaches for handling online updates.

3.2.1 Basic Concepts and Focus of the Study

Data Warehouse. Our study is motivated by large analytical data warehouses such as those characterized in the XLDB'07 report (Becla and Lim, 2008). There is typically a front-end operational (e.g., OLTP) system that produces the updates for the back-end analytical data warehouse. The data warehouse can be very large (e.g., petabytes), and does not fit in main memory.

Query Pattern. Large analytical data warehouses often observe "highly unpredictable query loads", as described in (Becla and Lim, 2008). Most queries involve "summary or aggregative queries spanning large fractions of the database". As a result, table range scans are frequently used in the queries. Therefore, we focus on table range scans as the optimization target: preserving the nice sequential data access patterns of table range scans in the face of online updates. The evaluation uses the TPC-H benchmark, which is a decision support benchmark with emphasis on ad-hoc queries.

Record Order. In row stores, records are often stored in primary key order (with clustered indexes). In addition, in DW applications, records are often generated ordered by the primary key. This property, called *implicit clustering*, allows to build access methods with specialized optimizations (Jagadish et al., 1997; Moerkotte, 1998). In column stores (that support online updates), the attributes of a record are aligned in separate columns allowing retrieval using a position value (RID) (Héman et al., 2010).[3] We assume that range scans provide data in the order of primary keys in row stores, and in the order of RIDs in column stores. Whenever primary keys in row stores and RIDs in column stores can be handled similarly, we use "key" to mean both.

Updates. Following prior work on differential updates (Héman et al., 2010), we optimize for incoming updates of the following forms: (i) inserting a record given its key; (ii) deleting a record given its key; or (iii) modifying the field(s) of a record to specified new value(s) given its key.[4,5] We call these updates *well-formed* updates. Data in large analytical data warehouses are often "write-once, read-many" (Becla and Lim, 2008), a special case of well-formed updates. Note that well-formed updates do not require reading existing data warehouse data. This type of updates is typical for Key-Value stores as well, where data are organized based on the key

---

[3] Following prior work (Héman et al., 2010), we focus on a single sort order for the columns of a table. We discuss how to support multiple sort orders in Section 3.6.

[4] We follow prior work on column stores to assume that the RID of an update is provided (Héman et al., 2010). For example, if updates contain sort keys, RIDs may be obtained by searching the (in-memory) index on sort keys.

[5] A modification that changes the key is treated as a deletion given the old key followed by an insertion given the new key.

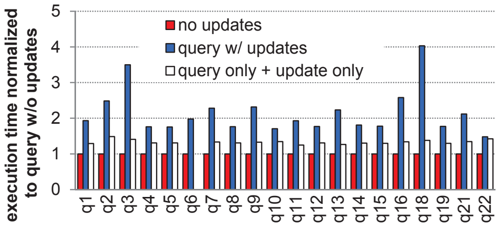


Figure 3.3: TPC-H queries with random updates on a row store.

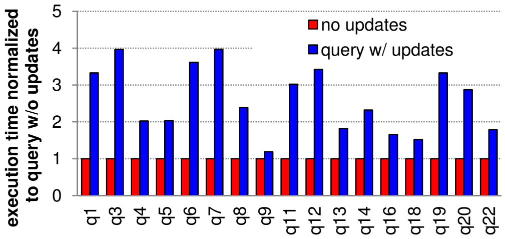


Figure 3.4: TPC-H queries with emulated random updates on a column store.

and the knowledge about the values is moved to the application. In such applications, most of the updates are "well-formed" and MaSM can offer significant benefits. In contrast to "well-formed updates", general transactions can require an arbitrary number of reads and writes. This distinction is necessary because the reads in general transactions may inherently require I/O reads to the main data and thus interfere with the sequential data access patterns in table range scans. For well-formed updates, our goal is to preserve range scan performance as if there were no online updates. For general transactions involving both reads and updates, we provide correct functionality, while achieving comparable or better performance than conventional online update handling, which we discuss in Section 3.5.2.

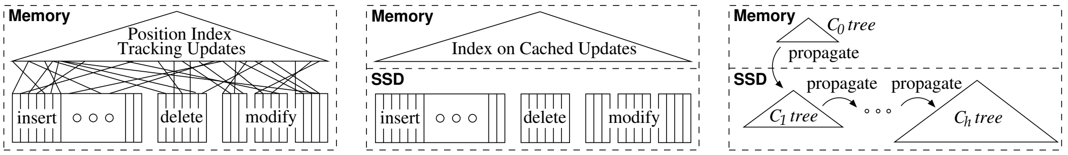
3.2.2 Conventional Approach: In-Place Updates

In order to clarify the impact of online random updates in analytic workloads, we execute TPC-H queries on both a commercial row-store *DBMS R* and a commercial column-store *DBMS C* while running online in-place updates.[6] The TPC-H scale factor is 30. We make sure that the database on disk is much larger than the allocated memory buffer size. We were able to perform concurrent updates and queries on the row store. However, the column store supports only offline updates, i.e., without concurrent queries. We recorded the I/O traces of offline updates, and when running queries on the column store, we use a separate program to replay the I/O traces outside of the DBMS to emulate online updates. During replay, we convert all I/O writes to I/O reads so that we can replay the disk head movements without corrupting the database.

Figure 3.3 compares the performance of TPC-H queries with no updates (first bar) and queries with online updates (second bar) on *DBMS R*. The third bar shows the sum of the first bar and the time for applying the same updates offline. Each cluster is normalized to the execution time of the first bar. Disk traces show sequential disk scans in all queries. As shown in Figure 3.3, queries with online updates see 1.5–4.1x slowdowns (2.2x on average), indicating significant performance degradation because of the random accesses of online updates. Moreover, the second bar is significantly higher than the third bar in most queries (with an average 1.6x extra slowdown). This shows that the increase in query response time is a result of not only having two workloads

---

[6] We were able to run 20 TPC-H queries on *DBMS R* and 17 TPC-H queries on *DBMS C*. The rest of the queries either do not finish in 24 hours, or are not supported by the DBMS.



(a) In-memory IU (b) Extending IU to SSDs (c) Applying LSM

Figure 3.5: Extending prior proposals of Indexed Updates (IU) to SSDs: (a) in-memory indexed updates (PDT (Héman et al., 2010)), (b) extending IU to SSDs, and (c) applying LSM (O'Neil et al., 1996) to IU.

executing concurrently, but also the interference between the two workloads. Figure 3.4 shows a similar comparison for the column store *DBMS C*. Compared with queries with no updates, running in-place updates online slows down the queries by 1.2–4.0x (2.6x on average).

3.2.3 Prior Proposals: Indexed Updates (IU)

Differential updates is the state-of-the-art technique for reducing the impact of online updates (Héman et al., 2010; Stonebraker et al., 2005; Jagadish et al., 1997). While the basic idea is straightforward (as described in Section 3.1), the efforts of prior work and this chapter are on the data structures and algorithms for efficiently implementing differential updates.

In-Memory Indexed Updates. Prior proposals maintain the cache for updates in main memory and build indexes on the cached updates (Héman et al., 2010; Stonebraker et al., 2005), which we call Indexed Updates (IU). Figure 3.5(a) shows the state-of-the-art IU proposal, Positional Delta Tree (PDT) designed for column stores (Héman et al., 2010). PDT caches updates in an insert table, a delete table, and a modify table per database attribute. It builds a positional index on the cached updates using RID as the index key. Incoming updates are appended to the relevant insert/delete/modify tables. During query processing, PDT looks up the index with RIDs to retrieve relevant cached updates. Therefore, the PDT tables are accessed in a random fashion during a range scan on the main data. Migration of the updates is handled by creating a separate copy of the main data, then making the new copy available when migration completes. Note that this requires twice as much data storage capacity as the main data size.

Problems of Directly Extending IU to SSDs. As discussed in Section 3.1, we aim to develop an SSD-based differential update solution that achieves the five design goals. To start, we consider directly extending IU to SSDs. As shown in Figure 3.5(b), the cached updates in insert/delete/modify tables are on SSDs. In order to *avoid random SSD writes*, incoming updates should be appended to these tables. For the same reason, ideally, the index is placed in memory because it sees a lot of random writes to accommodate incoming updates. Note that the index may consume a large amount of main memory, reducing the SSDs' benefit of saving memory footprint. We implemented this ideal-case IU following the above considerations (ignoring any memory footprint requirement). However, real-machine experiments show up to 3.8x query slowdowns even for this ideal-case IU (Section 3.5.2). We find that the slowdown is because the insert/delete/modify tables are randomly read during range scan operations. This is wasteful as an entire SSD page has to be read and discarded for retrieving a single update entry.

Problems of Applying LSM to IU. The log-structured merge-tree (LSM) is a disk-based index structure designed to support a high rate of insertions (O'Neil et al., 1996). An LSM consists of multiple levels of trees of increasing sizes. PDT employs the idea of multiple levels of trees to support snapshot isolation in memory (Héman et al., 2010). The stepped-merge algorithm (Jagadish et al., 1997) is based, as well, on the principle of storing updates in memory and organize them in progressively larger sorted runs, merging them eventually into a B^+-Tree. Here, we consider the feasibility of combining LSM and IU as an SSD-based solution.

As shown in Figure 3.5(c), LSM keeps the smallest C_0 tree in memory, and C_1, ..., C_h trees on SSDs, where $h \geq 1$. Incoming updates are first inserted into C_0, then gradually propagate to the SSD-resident trees. There are asynchronous rolling propagation processes between every adjacent pair (C_i, C_{i+1}) that (repeatedly) sequentially visit the leaf nodes of C_i and C_{i+1}, and move entries from C_i to C_{i+1}. This scheme avoids many of IU's performance problems. Random writes can be avoided by using large sequential I/Os during propagation. For a range scan query, it performs corresponding index range scans on every level of LSM, thereby avoiding wasteful random I/Os as in the above ideal-case IU.

Unfortunately, *LSM incurs a large amount of writes per update entry, violating the third design goal*. The additional writes arise in two cases: (i) An update entry is copied h times from C_0 to C_h; and (ii) the propagation process from C_i to C_{i+1} rewrites the old entries in C_{i+1} to SSDs once per round of propagation. According to (O'Neil et al., 1996), in an optimal LSM, the sizes of the trees form a geometric progression. That is, $size(C_{i+1})/size(C_i) = r$, where r is a constant parameter. It can be shown that in LSM the above two cases introduce about $r+1$ writes per update for levels $1, \ldots, h-1$ and $(r+1)/2$ writes per update for level h. As an example, with 4GB SSD space and 16MB memory (which is our experimental setting in Section 3.5.1), we can compute that a 2-level ($h = 1$) LSM writes every update entry ≈ 128 times. The optimal LSM that minimizes total writes has $h = 4$ and it writes every update entry ≈ 17 times! In other words, compared to a scheme that writes every update entry once, applying LSM on an SSD reduces its lifetime 17 fold (e.g., from 3 years to 2 months). For a more detailed comparison of the amount of writes performed by LSM and MaSM see Section 3.4.5.

3.3 MaSM Design

In this section, we propose MaSM (*materialized sort-merge*) algorithms for achieving the five design goals. We start by describing the basic ideas in Section 3.3.1. Then we present two MaSM algorithms: MaSM-2M in Section 3.3.2 and MaSM-M in Section 3.3.3. MaSM-M halves the memory footprint of MaSM-2M but incurs extra SSD writes. In Section 3.3.4, we generalize these two algorithms into a MaSM-αM algorithm, allowing a range of trade-offs between memory footprint and SSD writes by varying α. After that, we discuss a set of optimizations in Section 3.3.5, and describe transaction support in Section 3.3.6. Finally, we analyze the MaSM algorithms in terms of the five design goals in Section 3.3.7.

3.3.1 Basic Ideas

Consider the operation of merging a table range scan and the cached updates. For every record retrieved by the scan, it finds and applies any matching cached updates. Records without updates or new insertions must be returned too. Essentially, this is an outer join operation on the record key (primary key/RID).

Among various join algorithms, we choose to employ a sort-based join that sorts the cached updates and merges the sorted updates with the table range scan. This is because the most efficient joins are typically sort-based or hash-based, but hash-based joins have to perform costly I/O partitioning for the main data. The sort-based join also preserves the record order in table range scans, which allows hiding the implementation details of the merging operation from the query optimizer and upper-level query operators.

To reduce memory footprint, we keep the cached updates on SSDs and perform external sorting of the updates; two-pass external sorting requires $M = \sqrt{\|SSD\|}$ pages in memory to sort $\|SSD\|$ pages of cached updates on SSDs. However, external sorting may incur significant overhead for generating sorted runs and merging them. We exploit the following two ideas to reduce the overhead. First, we observe that a query should see all the cached updates that a previous query has seen. Thus, we materialize sorted runs of updates, deleting the generated runs only after update migration. This amortizes the cost of sorted run generation across many queries. Second, we would like to prune as many irrelevant updates to the current range scan query as possible. Because materialized runs are read-only, we can create a simple, read-only index, called *a run index*. A run index is a compact tree based index which records the smallest key (primary key/RID) for every fixed number of SSD pages in a sorted run. It is created after the sorted runs are formed, and, hence, it does not need to support insertion/deletion. Then we can search the query's key range in the run index to retrieve only those SSD pages that fall in the range. We call the algorithm combining the above ideas the *Materialized Sort-Merge (MaSM)* algorithm.

The picture of the above design is significantly complicated by the interactions among incoming updates, range scans, and update migrations. For example, sharing the memory buffer between updates and queries makes it difficult to achieve a memory footprint of M. In-place migrations may conflict with ongoing queries. Concurrency control and crash recovery must be re-examined. In the following, we first present a simple MaSM algorithm that requires $2M$ memory and a more sophisticated algorithm that reduces the memory requirement to M, then generalize them into an algorithm requiring αM memory. We propose a timestamp-based approach to support in-place migrations and ACID properties.

3.3.2 MaSM-2M

Figure 3.6 illustrates MaSM-2M, which allocates M pages to cache recent updates in memory and (up to) M pages for supporting a table range scan. Incoming (well-formed) updates are inserted into the in-memory buffer. When the buffer is full, MaSM-2M flushes the in-memory updates and creates a materialized sorted run of size M on the SSD. There are at most M materialized sorted runs since the capacity of the SSD is M^2. For a table range scan, MaSM-2M allocates one page in memory for scanning every materialized sorted run—up to M pages. It builds a Volcano-style (Graefe, 1994) operator tree to merge main data and updates, replacing the original `Table_range_scan` operator in query processing. We describe the detailed algorithm in Figure 3.7.

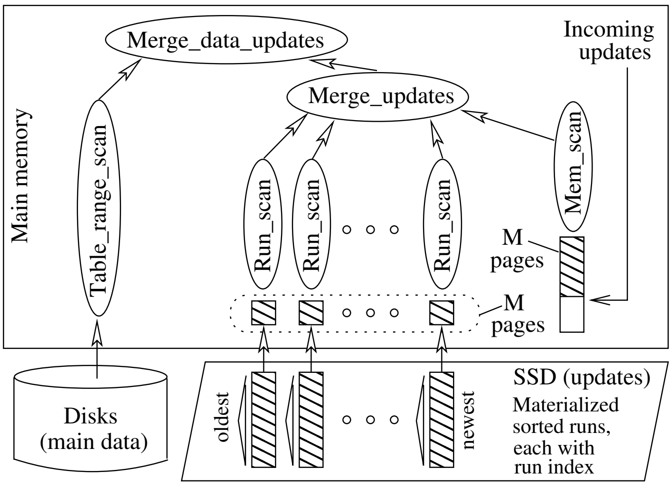


Figure 3.6: Illustrating the MaSM algorithm using $2M$ memory.

Timestamps. We associate every incoming update with a timestamp, which represents the commit time of the update. Every query is also assigned a timestamp. We ensure that a query can only see earlier updates with smaller timestamps. Moreover, we store in every database page the timestamp of the last update applied to the page, for supporting in-place migration. To do this, we reuse the Log Sequence Number (LSN) field in the database page. This field is originally used in recovery processing to decide whether to perform a logged redo operation on the page. However, in the case of MaSM, the LSN field is not necessary because recovery processing does not access the main data, rather it recovers the in-memory buffer for recent updates.

Update Record. For an incoming update, we construct a record of the format `(timestamp, key, type, content)`. As discussed in Section 3.2.1, table range scans output records in key order, either the primary key in a row store or the RID in a column store, and well-formed updates contain the key information. The `type` field is one of `insert/delete/modify/replace`; `replace` represents a deletion merged with a later insertion with the same key. The `content` field contains the rest of the update information: for `insert/replace`, it contains the new record except for the key; for `delete`, it is null; for `modify`, it specifies the attribute(s) to modify and the new value(s). During query processing, a `getnext` call on `Merge_data_updates` incurs `getnext` on operators in the subtree. `Run_scan` and `Mem_scan` scan the associated materialized sorted runs and the in-memory buffer, respectively. The `(begin key, end key)` of the range scan is used to narrow down the update records to scan. `Merge_updates` merges multiple streams of sorted updates. For updates with the same key, it merges the updates correctly. For example, the `content` fields of multiple modifications are merged. A deletion followed by an insertion changes the `type` field to `replace`. The operator `Merge_data_updates` performs the desired outer-join like merging operation of the data records and the update records.

Online Updates and Range Scan. Usually, incoming updates are appended to the end of the update buffer, and therefore do not interfere with ongoing `Mem_scan`. However, flushing the update buffer must be handled specially. MaSM records a flush timestamp with the update buffer therefore `Mem_scan` can discover the flushing. When this happens, `Mem_scan` instantiates a `Run_scan` operator for the new materialized sorted run and replaces itself with the `Run_scan` in the operator tree. The update buffer must be protected by latches (mutexes). To reduce latching overhead, `Mem_scan` retrieves multiple update records at a time.

Incoming Update. The steps to process an incoming update are:

```
if In-memory buffer is full then
    Sort update records in the in-memory buffer in key order;
    Build a run index recording (begin key, SSD page);
    Create a new materialized sorted run with run index on SSD;
    Reset the in-memory buffer;
end if
Append the incoming update record to the in-memory buffer;
```

Table Range Scan. MaSM-2M constructs a Volcano-style query operator tree to replace the `Table_range_scan` operator:

```
Instantiate a Run_scan operator per materialized sorted run using the run index to narrow
    down the SSD pages to retrieve;
Sort the in-memory buffer for recent update records;
Instantiate a Mem_scan operator on the in-memory buffer and locate the begin and end
    update records for the range scan;
Instantiate a Merge_updates operator as the parent of all the Run_scan operators and the
    Mem_scan operator;
Instantiate a Merge_data_updates operator as the parent of the Table_range_scan
    and the Merge_updates;
return Merge_data_updates;
```

Figure 3.7: MaSM-2M algorithm

Multiple Concurrent Range Scans. When multiple range scans enter the system concurrently, each one builds its own query operator tree with distinct operator instances, and separate read buffers for the materialized sorted runs. `Run_scan` performs correctly because materialized sorted runs are read-only. On the other hand, the in-memory update buffer is shared by all `Mem_scan` operators, which sort the update buffer then read sequentially from it. For reading the buffer sequentially, each `Mem_scan` tracks `(key, pointer)` pairs for the `next` position and the `end_range` position in the update buffer. To handle sorting, MaSM records a sort timestamp with the update buffer whenever it is sorted. In this way, `Mem_scan` can detect a recent sorting. Upon detection, `Mem_scan` adjusts the pointers of the `next` and `end_range` positions by searching for the corresponding keys. There may be new update records that fall between the two pointers after sorting. `Mem_scan` correctly filters out the new update records based on the query timestamp.

In-Place Migration of Updates Back to Main Data. Migration is implemented by performing a full table scan where the scan output is written back to disks. Compared to a normal table range scan, there are two main differences. First, the `(begin key, end key)` is set to cover the entire range. Second, `Table_range_scan` returns a sequence of data pages rather than a sequence of records. `Merge_data_updates` applies the updates to the data pages in the database buffer pool, then issues (large sequential) I/O writes to write back the data pages. For compressed data chunks in a column store, the data are uncompressed in memory, modified, re-compressed, and written back to disks. The primary key index or the RID position index is updated along with every data page. Here, by in-place migration, we mean two cases: (i) new data pages overwrite the old pages with the same key ranges; or (ii) after writing a new chunk of data, the corresponding old data chunk is deleted so that its space can be used for writing other new data chunks.

The migration operation is performed only when the system is under low load or when the size of cached updates is above a pre-specified threshold. When one of the conditions is true, MaSM logs the current timestamp t and the IDs of the set R of current materialized sorted runs in the redo log, and spawns a migration thread. The thread waits until all ongoing queries earlier than t complete, then migrates the set R of materialized runs. When migration completes, it logs a migration completion record in the redo log then deletes the set R of materialized runs. Note that any queries that arrive during migration are evaluated correctly without additional delays. The table scan performed by the migration thread serve them correctly by delivering the data with the appropriate updates. On the other hand, the migration process begins only after all queries with starting time earlier than the selected migration timestamp t have finished.

3.3.3 MaSM-M

Figure 3.8 illustrates MaSM-M, the MaSM algorithm using M memory. There are two main differences between MaSM-M and MaSM-2M. First, compared to MaSM-2M, MaSM-M manages memory more carefully to reduce its memory footprint to M pages. S of the M pages, called update pages, are dedicated to buffering incoming updates in memory. The rest of the pages, called query pages, are mainly used to support query processing. Second, materialized sorted runs have different sizes in MaSM-M. Since the query pages can support up to only $M-S$ materialized sorted runs, the algorithm has to merge multiple smaller materialized sorted runs into larger materialized sorted runs. We call the sorted runs that are directly generated from the in-memory buffer *1-pass* runs, and those resulted from merging 1-pass runs *2-pass* runs.

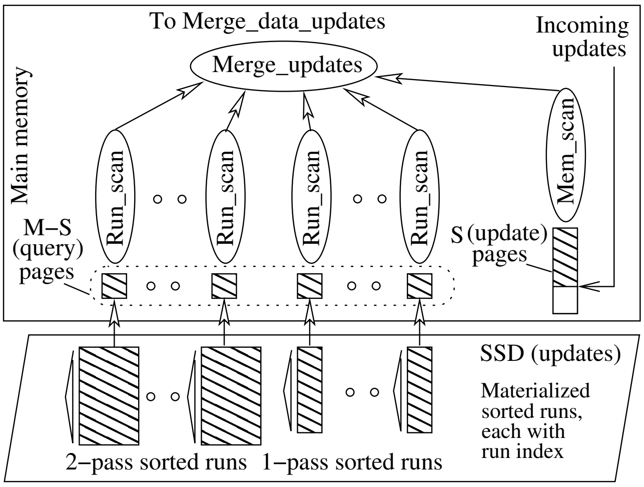


Figure 3.8: Illustrating MaSM algorithm using M memory.

All *1-pass* sorted runs comprised of S pages each. Since *1-pass* sorted runs have the same size, when generating *2-pass* runs we merge N contiguous *1-pass* sorted runs. Any other approach would incur a higher number of writes on the device leading to undesired increase in the write amplification of the MaSM algorithm. Replacement Selection (Knuth, 1998) offers less total number writes and better merge performance, but breaks the correct operation of the migration process discussed in Section 3.3.2. The *1-pass* sorted runs are created sequentially and, hence, they have disjoint timestamp sets. This property is important during migration and crash recovery in order to guarantee that there is a timestamp t before which, all updates are taken into account when the migration is finished. Using Replacement Selection would generate sorted runs with timestamp overlaps, making it impossible for the migration process to complete correctly.

Figure 3.9 presents the pseudo code of MaSM-M. Algorithm parameters are summarized in Table 3.1. Incoming updates are cached in the in-memory buffer until the buffer is full. At this moment, the algorithm tries to steal query pages for caching updates if they are not in use (Lines 2–3). The purpose is to create *1-pass* runs as large as possible for reducing the need for merging multiple runs. When query pages are all in use, the algorithm materializes a *1-pass* sorted run with run index on SSDs (Line 5).

The table range scan algorithm consists of three parts. First, Lines 1–4 create a *1-pass* run if the in-memory buffer contains at least S pages of updates. Second, Lines 5–8 guarantee that the number of materialized sorted runs is at most $M-S$, by (repeatedly) merging the N earliest 1-pass sorted runs into a single 2-pass sorted run until $K_1 + K_2 \le M - S$. Note that the N earliest 1-pass runs are adjacent in time order, and thus merging them simplifies the overall MaSM operation when merging updates with main data. Since the size of a 1-pass run is at least S pages, the size of a 2-pass sorted run is at least NS pages. Finally, Lines 9–14 construct the query operator tree. This part is similar to MaSM-2M.

Like MaSM-2M, MaSM-M handles concurrent range scans, updates, and in-place migration using the timestamp approach.

Minimizing SSD Writes for MaSM-M. We choose the S and N parameters to minimize SSD writes for MaSM-M. Theorem 3.1 computes the bound of the number of SSD writes that MaSM incurs. In fact, it is a corollary of a more general result shown in the next section (Theorem 3.2).

Incoming Updates:

```
if In-memory buffer is full then
    if At least one of the query pages is not used then
        Steal a query page to extend the in-memory buffer;
    else
        Create a 1-pass materialized sorted run with run index from the updates in the in-memory buffer;
        Reset the in-memory buffer to have $S$ empty pages;
    end if
end if
Append the incoming update record to the in-memory buffer;
```

Table Range Scan Setup:

```
if In-memory buffer contains at least $S$ pages of updates then
    Create a 1-pass materialized sorted run with run index from the updates in the in-memory buffer;
    Reset the in-memory buffer to have $S$ empty update pages;
end if
while $K_1 + K_2 > M - S$ do {merge 1-pass runs}
    Merge $N$ earliest adjacent 1-pass runs into a 2-pass run;
    $K_1 = K_1 - N$; $K_2$++;
end while
Instantiate a Run_scan operator per materialized sorted run using the run index to narrow down SSD pages to retrieve;
if In-memory buffer is not empty then
    Sort the in-memory buffer and create a Mem_scan operator;
end if
Instantiate a Merge_updates operator as the parent of all the Run_scan operators and the Mem_scan operator;
Instantiate a Merge_data_updates operator as the parent of the Table_range_scan and the Merge_updates;
return Merge_data_updates;
```

Figure 3.9: MaSM-M algorithm

Table 3.1: Parameters used in the MaSM-M algorithm.

$\|SSD\|$	SSD capacity (in pages), $\|SSD\| = M^2$
M	memory size (in pages) allocated for MaSM-M
S	memory buffer (in pages) allocated for incoming updates
K_1	number of 1-pass sorted runs on SSDs
K_2	number of 2-pass sorted runs on SSDs
N	merge N 1-pass runs into a 2-pass run, $N \leq M - S$

Theorem 3.1. *The MaSM-M algorithm minimizes the number of SSD writes in the worst case when* $S_{opt} = 0.5M$ *and* $N_{opt} = 0.375M + 1$. *The average number of times that MaSM-M writes every update record to SSD is* $1.75 + \frac{2}{M}$.

3.3.4 MaSM-αM

We generalize the MaSM-M algorithm to a MaSM-αM algorithm that uses αM memory, where $\alpha \in [\frac{2}{M^{\frac{1}{3}}}, 2]$. The algorithm is the same as MaSM-M except that the total allocated memory size is αM pages. The lower bound on α ensures that the memory is sufficiently large to make 3-pass sorted runs unnecessary. MaSM-M is a special case of MaSM-αM when $\alpha = 1$, and MaSM-2M is a special case of MaSM-αM when $\alpha = 2$. Similar to Theorem 3.1, we can obtain the following theorem for minimizing SSD writes for MaSM-αM.

Theorem 3.2. *Let* $\alpha \in [\frac{2}{M^{\frac{1}{3}}}, 2]$. *The MaSM-$\alpha$M algorithm minimizes the number of SSD writes in the worst case when* $S_{opt} = 0.5\alpha M$ *and* $N_{opt} = \frac{1}{\lfloor \frac{4}{\alpha^2} \rfloor}(\frac{2}{\alpha} - 0.5\alpha)M + 1$. *The average number of times that MaSM-αM writes every update record to SSD is roughly* $2 - 0.25\alpha^2$.

We can verify that MaSM-M incurs roughly $2 - 0.25 * 1^2 = 1.75$ writes per update record, while MaSM-2M writes every update record once ($2 - 0.25 * 2^2 = 1$).

Theorem 3.2 shows the trade-off between memory footprint and SSD writes for a spectrum of MaSM algorithms. At one end of the spectrum, MaSM-2M achieves minimal SSD writes with 2M memory. At the other end of the spectrum, one can achieve a MaSM algorithm with very small memory footprint ($2M^{\frac{2}{3}}$) while writing every update record at most twice.

3.3.5 Further Optimizations

Granularity of Run Index. Because run indexes are read-only, their granularity can be chosen flexibly. For example, suppose the page size of the sorted runs on SSDs is 64KB. Then we can keep the begin key for a coarser granularity, such as one key per 1MB, or a finer granularity, such as one key per 4KB. The run index should be cached in memory for efficient accesses (especially if there are a lot of small range scans). Coarser granularity saves memory space, while finer granularity makes range scans on the sorted run more precise. The decision should be made based on the query workload. The former is a good choice if very large ranges are typical, while the latter should be used if narrower ranges are frequent. For indexing purpose, one can keep a 4-byte prefix of the actual key in the run index. Therefore, the fine-grain indexes that keep 4 bytes for every 4KB updates are $\|SSD\|/1024$ large. If keeping 4 bytes for every 1MB updates, the total run index size is $\|SSD\|/256K$ large. In both cases the space overhead on SSD is very small: $\approx 0.1\%$ and $\approx 4 \cdot 10^{-4}\%$ respectively. Note that there is no reason to have finer granularity than one value (4 bytes in our setup) every 4KB of updates because this is the access granularity as well. Hence, the run index overhead is never going to be more than 0.1%.

Handling Skews in Incoming Updates. When updates are highly skewed, there may be many duplicate updates (i.e., updates to the same record). The MaSM algorithms naturally handle skews: When generating a materialized sorted run, the duplicates can be merged in memory as long as the merged update records do not affect the correctness of concurrent range scans. That is, if two update records with timestamp $t1$ and $t2$ are to be merged, there should not be any concurrent range scans with timestamp t, such that $t1 < t \leq t2$. In order to further reduce duplicates, one can compute statistics about duplicates at the range scan processing time. If the benefits of removing duplicates outweigh the cost of SSD writes, one can remove all duplicates by generating a single materialized sorted run from all existing runs.

Improving Migration. There are several ways to improve the migration operation. First, similar to several techniques to share scans (coordinated scans (Fernandez, 1994), circular scans (Harizopoulos et al., 2005), cooperative scans (Zukowski et al., 2007)), we can combine the migration with a table scan query in order to avoid the cost of performing a table scan for migration purposes only. Hence, any queries that arrives during the migration process can be attached to the migration table scan and use the newly merged data directly as their output. In case they did not attach at the very beginning of the migration process, the queries spawn a new range scan from the beginning of the fact table until the point they actually started from. Since the data residing on disk is the same in the two parts of the query, i.e., the version of the data after the latest migration, a simple row id is enough to signify the ending of the second range scan. Second, one can migrate a portion (e.g., every $1/10$ of table range) of updates at a time to distribute the cost across multiple operations, shortening any delays for queries arriving during an ongoing migration. To do this, each materialized sorted run records the ranges that have already been migrated and the ranges that are still active.

3.3.6 Transaction Support

The lifetime of an update. A new update, U, arriving in the system is initially logged on the transactional log and, subsequently, stored in the in-memory buffer. The transactional log guarantees that in an event of a crash, the updates stored in-memory will not be lost. When the buffer gets full it is sorted and it is sequentially written as a 1-pass materialized sorted run on the SSD. Once this step is completed the record for U on the transactional log is not needed, because MaSM guarantees the durability of U on the SSD. Hence, the transactional log can be truncated as often as the in-memory buffer is flushed to flash. For MaSM-αM and $\alpha < 2$ (and, hence, MaSM-M as well) 1-pass sorted runs are merged and written a second time on the SSD on 2-pass materialized sorted runs. For MaSM-2M this step is skipped. When the SSD free capacity is less than a given threshold, the migration process is initiated. During this process, all updates are applied on the main data during a full table scan. Each data page has its timestamp updated to be equal to the timestamp of the last added update. The migration process is protected with log records that make sure that in an event of a crash the process will start over. When the migration process completes correctly, the sorted-runs including migrated updates are removed from the SSD and the relevant updates are now part only of the main data.

Serializability among Individual Queries and Updates. By using timestamps, MaSM algorithms guarantee that queries see only earlier updates. In essence, the timestamp order defines a total serial order, and thus MaSM algorithms guarantee serializability among individual queries and updates.

Supporting Snapshot Isolation for General Transactions. In snapshot isolation (Berenson et al., 1995), a transaction works on the snapshot of data as seen at the beginning of the transaction. If multiple transactions modify the same data item, the first committer wins while the other transactions abort and roll back. Note that snapshot isolation alone does not solve the online updates problem in data warehouses. While snapshot isolation removes the logical dependencies between updates and queries so that they may proceed concurrently, the physical interferences between updates and queries present major performance problems. Such interferences are the target of MaSM algorithms.

Similar to prior work (Héman et al., 2010), MaSM can support snapshot isolation by maintaining for every ongoing transaction a small private buffer for the updates performed by the transaction. (Note that such private buffers may already exist in the implementation of snapshot isolation.) A query in the transaction has the timestamp of the transaction start time so that it sees only the snapshot of data at the beginning of the transaction. To incorporate the transaction's own updates, we can instantiate a `Mem_scan` operator on the private update buffer, and insert this operator in the query operator tree in Figure 3.6 and 3.8. At commit time, if the transaction succeeds, we assign the commit timestamp to the private updates and copy them into the global in-memory update buffer.

Supporting Locking Schemes for General Transactions. Shared (exclusive) locks are used to protect reads (writes) in many database systems. MaSM can support locking schemes as follows. First, for an update, we ensure that it is globally visible only after the associated exclusive lock is released. To do this, we allocate a small private update buffer per transaction (similar to snapshot isolation), and cache the update in the private buffer. Upon releasing the exclusive lock that protects the update, we assign the current timestamp to the update record and append it to

MaSM's global in-memory update buffer. Second, we assign the normal start timestamp to a query so that it can see all the earlier updates.

For example, two phase locking is correctly supported. In two phase locking, two conflicting transactions A and B are serialized by the locking scheme. Suppose A happens before B. Our scheme makes sure that A's updates are made globally visible at A's lock releasing phase, and B correctly see these updates.

The In-Memory Update Buffer. The incoming updates are first stored in an in-memory buffer before they are sorted and written on the SSD. During query execution (one or more concurrent queries) this buffer can be read by every active query without any protection required. In fact, even in the presence of updates only minimal protection is needed because no query will ever need to read updates made by subsequent transactions. Hence, the buffer is protected for consistency with a simple latch. However, when updates are issued concurrently the in-memory buffer is bound to get full. In this case the buffer is locked, sorted and flushed to the SSD. During the in-memory buffer flush MaSM delays the writes. A different approach is to have a second buffer and switch the pointers between the two buffers when one gets full, sorted and flushed to the SSD.

Crash Recovery. Typically, MaSM needs to recover only the in-memory update buffer for crash recovery. This can be easily handled by reading the database redo log for the update records. It is easy to use update timestamps to distinguish updates in memory and updates on SSDs. In the rare case, the system crashes in the middle of an update migration operation. To detect such cases, MaSM records the start and the end of an update migration in the log. Note that we do not log the changes to data pages in the redo log during migration, because MaSM can simply redo the update migration during recovery processing; by comparing the per-data-page timestamps with the per-update timestamps, MaSM naturally determines whether updates should be applied to the data pages. Cached updates are removed only after the migration succeeds. Therefore, this process is idempotent, so in an event of multiple crashes it can still handle correctly the updates.

The primary key index or RID position index is examined and updated accordingly.

3.3.7 Achieving The Five Design Goals

As described in Sections 3.3.2–3.3.6, it is clear that the MaSM algorithms perform *no random SSD writes* and provide *correct ACID support*. We analyze the other three design goals in the following.

Low Overhead for Table Range Scan Queries. Suppose that updates are uniformly distributed across the main data. If the main data size is $\|Disk\|$ pages, and the table range scan query accesses R disk pages, then MaSM-αM reads $max(R\frac{\|SSD\|}{\|Disk\|}, 0.5\alpha M)$ pages on SSDs. This formula has two parts. First, when R is large, run indexes can effectively narrow down the accesses to materialized sorted runs, and therefore MaSM performs $(R\frac{\|SSD\|}{\|Disk\|})$ SSD I/Os, proportional to the range size. Compared to reading the disk main data, MaSM reads fewer bytes from SSD, as $\frac{\|SSD\|}{\|Disk\|}$ is 1%–10%. Therefore, the SSD I/Os can be completely overlapped with the table range scan on main data, leading to very low overhead. A detailed analysis of the impact of the SSD I/Os is presented both through an analytical model in Section 3.4.4 and through experimentation in Section 3.5.2.

Second, when the range R is small, MaSM-αM performs at least one I/O per materialized sorted run: the I/O cost is bounded by the number of materialized sorted runs (up to $0.5\alpha M$). (Our experiments in Section 3.5.2 see 128 sorted runs for 100GB data.) Note that SSDs can support 100x–1000x random 4KB reads per second compared to disks (Intel, 2009). Therefore, MaSM can overlap most latencies of the $0.5\alpha M$ random SSD reads with the small range scan on disks, achieving low overhead.

Bounded memory footprint. Theorem 3.2 shows that our MaSM algorithms use $[2M^{\frac{2}{3}}, 2M]$ pages of memory, where $M = \sqrt{\|SSD\|}$.

Bounded number of total writes on SSD. MaSM has a limited effect in the device lifetime by bounding the number of writes per update according to Theorem 3.2. All MaSM algorithms use fewer than 2 SSD writes per update, with MaSM-2M using only 1 write per update.

Efficient In-Place Migration. MaSM achieves in-place update migration by attaching timestamps to updates, queries, and data pages as described in Section 3.3.2.

We discuss two aspects of migration efficiency. First, MaSM performs efficient sequential I/O writes in a migration. Because update cache size is non-trivial (1%–10%) compared to main data size, it is likely that there exist update records for every data page. Compared to conventional random in-place updates, MaSM can achieve orders of magnitude higher sustained update rate, as is shown in Section 3.5.2. Second, MaSM achieves low migration frequency with a small memory footprint. If SSD page size is P, then MaSM-αM uses $F = \alpha MP$ memory capacity to support an SSD-based update cache of size $M^2P = \frac{F^2}{\alpha^2 P}$. Note that as the memory footprint doubles, the size of MaSM's update cache increases by a factor of 4, and the migration frequency decreases by a factor of 4, as compared to a factor of 2 with prior approaches that cache updates in memory (see Figure 3.1). For example, for MaSM-M, if $P = 64$KB, a 16GB in-memory update cache in prior approaches has the same migration overhead as just an $F = 32$MB in-memory buffer in our approach, because $F^2/(64\text{KB}) = 16$GB.

3.4 Analysis and Modeling of MaSM

In this section, we present the proofs of the theorems in Section 3.3 and provide an in-depth analysis of the behavior of the MaSM algorithm.

3.4.1 Theorems About MaSM Behavior

We start by proving Theorem 3.2 for the general case and then we show how to deduce Theorem 3.1.

Theorem 3.2. *Let* $\alpha \in [\frac{2}{M^{\frac{1}{3}}}, 2]$. *The MaSM-αM algorithm minimizes the number of SSD writes in the worst case when* $S_{opt} = 0.5\alpha M$ *and* $N_{opt} = \frac{1}{\lfloor \frac{4}{\alpha^2} \rfloor}(\frac{2}{\alpha} - 0.5\alpha)M + 1$. *The average number of times that MaSM-αM writes every update record to SSD is roughly* $2 - 0.25\alpha^2$.

Proof. Every update record is written at least once to a 1-pass run. Extra SSD writes occur when 1-pass runs are merged into a 2-pass run. We choose S and N to minimize the extra writes.

The worst case scenario happens when all the 1-pass sorted runs are of the minimal size S. In this case, the pressure for generating 2-pass runs is the greatest. Suppose all the 1-pass runs have size S and all the 2-pass runs have size NS. We must make sure that when the allocated SSD space is full, MaSM-αM can still merge all the sorted runs. Therefore:

$$K_1 S + K_2 NS = M^2 \tag{3.1}$$

and

$$K_1 + K_2 \leq \alpha M - S \tag{3.2}$$

Compute K_1 from (3.1) and plugging it into (3.2) yields:

$$K_2 \geq \frac{1}{N-1}(S - \alpha M + \frac{M^2}{S}) \tag{3.3}$$

When the SSD is full, the total extra writes is equal to the total size of all the 2-pass sorted runs:

$$ExtraWrites = K_2 NS \geq \frac{N}{N-1}(S^2 - \alpha MS + M^2) \tag{3.4}$$

To minimize $ExtraWrites$, we would like to minimize the right hand side and achieve the equality sign as closely as possible. The right hand side achieves the minimum when N takes the largest possible value and $S_{opt} = 0.5\alpha M$. Plug it into (3.3) and (3.4):

$$K_2 \geq \frac{1}{N-1}(\frac{2}{\alpha} - 0.5\alpha)M \tag{3.5}$$

$$ExtraWrites = 0.5\alpha M K_2 N \geq \frac{N}{N-1}(1 - 0.25\alpha^2)M^2 \tag{3.6}$$

Note that the equality signs in the above two inequalities are achieved at the same time. Given a fixed K_2, the smaller the N, the lower the $ExtraWrites$. Therefore, $ExtraWrites$ is minimized when the equality signs are held. We can rewrite $min(ExtraWrites)$ as a function of K_2:

$$min(ExtraWrites) = (0.5\alpha\frac{K_2}{M} + 1 - 0.25\alpha^2)M^2 \tag{3.7}$$

The global minimum is achieved with the smallest non-negative integer K_2. Since $N \leq \alpha M - S_{opt} = 0.5\alpha M$, we know $K_2 > \frac{4}{\alpha^2} - 1$ according to (3.5).

Therefore, $ExtraWrites$ achieves the minimum when $S_{opt} = 0.5\alpha M$, $K_{2,opt} = \lfloor\frac{4}{\alpha^2}\rfloor$, and $N_{opt} = \frac{1}{K_{2,opt}}(\frac{2}{\alpha} - 0.5\alpha)M + 1$. We can compute the minimum $ExtraWrites$ $\simeq (1 - 0.25\alpha^2)M^2$. In this setting, the average number of times that MaSM-αM writes every update record to SSD is roughly $2 - 0.25\alpha^2$. Moreover, since $K_2 \leq \alpha M - S_{opt}$ iff $\lfloor\frac{4}{\alpha^2}\rfloor \leq 0.5\alpha M$, we know that $\alpha \geq \frac{2}{M^{\frac{1}{3}}}$. □

Theorem 3.1. *The MaSM-M algorithm minimizes the number of SSD writes in the worst case when* $S_{opt} = 0.5M$ *and* $N_{opt} = 0.375M + 1$. *The average number of times that MaSM-M writes every update record to SSD is* $1.75 + \frac{2}{M}$.

Proof. When $\alpha = 1$, Theorem 3.2 yields $S_{opt} = 0.5M$ and $N_{opt} = \frac{1}{4}(2 - 0.5)M + 1 = \frac{3}{8}M + 1$. Moreover, equation 3.7 with $K_2 = \lfloor\frac{4}{\alpha^2}\rfloor = 4$ yields $ExtraWrites = (\frac{2}{M} + 1 – 0.25)M^2 = 0.75M^2 + 2M$. In this setting, the average number of times that MaSM-M writes every update record to SSD is $1 + (0.75M^2 + 2M)/M^2 = 1.75 + \frac{2}{M} \simeq 1.75$. □

3.4.2 SSD Wear vs. Memory Footprint

MaSM-αM specifies a spectrum of MaSM algorithms trading off SSD writes for memory footprint. By varying α from 2 to $\frac{2}{M^{\frac{1}{3}}}$, the memory footprint reduces from $2M$ pages to $2M^{\frac{2}{3}}$ pages, while the average times that an update record is written to SSDs increases from the (minimal) 1 to close to 2. In all MaSM algorithms, we achieve small memory footprint and low total SSD writes.

In Figure 3.10 we vary the value of α on the x-axis and depict the resulting number of writes per updates of the corresponding MaSM-αM algorithm. The figure shows the operating point of the MaSM-2M algorithm (on the rightmost part of the graph), the operating point of the MaSM-M (in the middle) and the point corresponding to the lower bound of the value of the parameter α, which is the smallest possible memory footprint allowing MaSM to avoid 3-pass external sorting. We see that the minimum value of α for 4GB SSD space, is $\alpha \approx 0.02$, leading to 100x less memory for twice the number of writes per update, compared to MaSM-2M. Therefore, one can choose the desired parameter α based on the requirements of the application and the capacities of the underlying SSD.

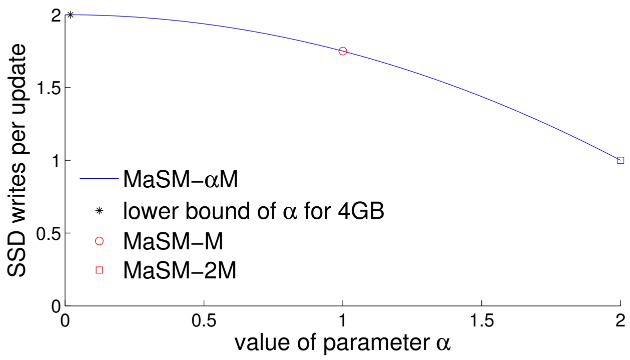


Figure 3.10: The tradeoff between SSD wear and memory footprint.

The write endurance of the enterprise-grade SLC (Single Level Cell) NAND Flash SSD, 32GB Intel X25E SSD, used in our experimentation is 10^5. Therefore, the Intel X25E SSD can support 3.2-petabyte writes, or 33.8MB/s for 3 years. MaSM writes on flash using large chunks, equal to an erase block for several SSDs, allowing the device to operate with minimal write amplification, close to 1. MaSM-2M writes SSDs once for any update record, therefore a single SSD can sustain up to 33.8MB/s updates for 3 years. MaSM-M writes about 1.75 times for an update record. Therefore, for MaSM-M, a single SSD can sustain up to 19.3MB/s updates for 3 years, or 33.8MB/s for 1 year and 8 months. This limit can be improved by using larger total SSD capacity: doubling SSD capacity doubles this bound.

Newer enterprise-grade NAND Flash SSDs, however, have switched to MLC (Multi Level Cell). The endurance of a MLC cell is typically one order of magnitude lower than SLC, 10^4, and the capacity of MLC NAND Flash SSDs about one order of magnitude higher, 200GB to 320GB. The lack of endurance in erase cycles is balanced out by the larger capacity and, as a result, the lifetime of an MLC SSD is in fact similar to an SLC SSD.

3.4.3 Memory Footprint vs. Performance

As discussed in Section 3.3.7, the I/O costs of small range scans are determined by the number of materialized sorted runs. For the extreme case of a point query, where a single page of data is retrieved, the merging process has to fetch one page of updates from every sorted run. In this case the response time is dominated by the time of reading from the SSD. In order to mitigate this bottleneck one has to reduce the number of sorted runs stored on the SSD.

For the family of MaSM-αM algorithms, the number of sorted runs stored on the SSD can be calculated using Equation 3.2. If we assume the optimal distribution of buffer pages as described in Theorem 3.2, i.e., $S_{opt} = \frac{\alpha \cdot M}{2}$, the equality holds for Equation 3.2 and we get the maximum number of sorted runs, N:

$$N = K_1 + K_2 = \alpha \cdot M - S_{opt} = \frac{\alpha \cdot M}{2} \tag{3.8}$$

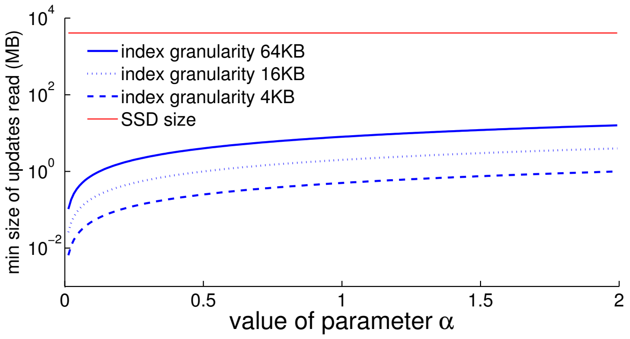


Figure 3.11: Memory footprint vs. short range performance.

A new tradeoff is formed: by increasing the allocated memory for the algorithm (i.e. by increasing α)[7] one increases the lower bound of the cost of reading updates from the SSD in the case of sort range scans. In fact, the lower bound of updates read in MB from the SSD is given by the product of the number of sorted runs, N, the SSD page size, P, and the index granularity fraction, g, which is defined as the granularity of the run index divided by the page size. Finally, $M = \sqrt{\|SSD\|}$ and α is the parameter showing the available memory to the MaSM-αM algorithm.

$$UpdatesRead = N \cdot g \cdot P = \frac{\alpha \cdot M \cdot g \cdot P}{2} \tag{3.9}$$

Equation 3.9 shows the relationship between the size of updates to be read and the value of the parameter α. Figure 3.11 shows how the lower bound of the size of fetched updates (y-axis in MB) changes as α changes (x-axis). Three blue lines are presented each one corresponding to index granularity 64KB, 16KB and 4KB respectively. The straight red line on the top of the graph corresponds to the overall size of the SSD which in our experimentation and analysis is 4GB and the SSD page size P is equal to 64KB. Higher values of α lead to higher I/O cost for fetching the updates, because a bigger percentage of the updates stored on SSD should be read. The above analysis is extended in the following section in order to fully understand how the overhead of

---

[7] Increasing α helps with SSD wear (Section 3.4.2) but it has adverse effects regarding short range queries performance. Hence, there is a tradeoff between SSD wear, allocated memory, and short range queries performance.

reading the updates affects the performance of the MaSM algorithms.

3.4.4 Modeling the I/O Time of Range Queries

In this section we present an analytical model for range queries, which considers the I/O time of reading updates from SSDs, and the I/O time of reading the main data. We model various aspects of the range queries, including relation size, disk capacity, SSD capacity, range size, and disk and SSD performance characteristics. The parameters used in the performance model, including the ones detailed in the discussion about the MaSM algorithms in Section 3.3.4 (see Table 3.1), are summarized in Table 3.2.

In Sections 3.3.7 and 3.4.3 it is argued that the lower bound for the cost of retrieving updates from SSD is given by the number of sorted runs. The cost of retrieving updates for larger ranges, however, is a function of the range of the relation we retrieve. More specifically, if we assume that the updates are uniformly distributed over the main data then the number of SSD pages retrieved is proportional to the range size of the query. Hence, the time needed to fetch the updates is given by Equation 3.10.

$$FetchUpdates = max(\|SSD\| \cdot r, N \cdot g) \cdot \frac{P}{FBW} \tag{3.10}$$

The size of the range, $r\%$ is expressed as a percentage of the total size of the relation (T). $\|SSD\|$ is the number of SSD pages devoted to the MaSM algorithm, g is the granularity of the run index as a fraction of SSD pages, which in turn have size P. The bandwidth that updates are fetched from the SSD, FBW, defines how fast MaSM can read updates. In Equation 3.10 we first calculate the number of pages containing updates to be read, which is equal to $max(\|SSD\|r, Ng)$, and then we take the worst case scenario regarding the I/O time per page.

MaSM overlaps SSD I/Os for retrieving updates with HDD I/Os for retrieving the main data. In order to model this behavior, we devise a simple HDD model. For a query that requires a range scan of $r\%$ of the main data (with total size T) the time needed to retrieve the data is given by Equation 3.11.

$$ReadMainData = s + \frac{r \cdot T}{DBW} \tag{3.11}$$

The seek time, s, plays an important role since it dominates the response time for short ranges. For large ranges, however, the most important factor is the disk bandwidth, DBW.

Achieving perfect I/O overlapping using MaSM. Using Equations 3.10 and 3.11 we compute

Table 3.2: The parameters of MaSM-M's performance model and their values.

T	The total size of the table	100 GB
s	The seek time of the main storage hard disk	4.17 ms, 3 ms, 2 ms
DBW	The sequential bandwidth offered by the main storage hard disk	77 MB/s, 200 MB/s, 800 MB/s
r	The percentage of the relation queried sequentially	$2^{-26} \ldots 2^{-1}, 1$
$\|SSD\|$	SSD capacity (in pages)	65536 pages
M	MaSM-M memory, $M = \sqrt{\|SSD\|}$	256 pages
P	SSD page size	64 KB
α	The α parameter, giving the amount of memory available to MaSM-αM	0.5, 1.0, 2.0
N	The number of sorted runs	64, 128, 256
g	The granularity fraction of the run index	1, 1/16
$ReadBW$	The nominal read bandwidth offered by the SSD	250 MB/s, 1500 MB/s
FBW	The random read bandwidth offered by the SSD when reading P-sized pages	193 MB/s, 578 MB/s

under which conditions the I/O for fetching updates can be entirely overlapped by the I/O needed to read the main data, i.e., *the formal condition for perfect I/O overlapping*:

$$ReadMainData \geq FetchUpdates \Rightarrow s + \frac{r \cdot T}{DBW} \geq max(\|SSD\| \cdot r, N \cdot g) \cdot (\frac{P}{FBW}) \Rightarrow$$

$$s \geq max(\|SSD\| \cdot r, N \cdot g) \cdot \frac{P}{FBW} - \frac{r \cdot T}{DBW} \tag{3.12}$$

Equation 3.12 has two branches, depending on whether $\|SSD\| \cdot r$ has higher value than $N \cdot g$ or not. When the query range is large enough to bring at least one page per sorted run (i.e., $\|SSD\| \cdot r \geq N \cdot g$) the condition becomes:

$$s \geq \frac{\|SSD\| \cdot r \cdot P}{FBW} - \frac{r \cdot T}{DBW} \Rightarrow s \geq r \cdot \left(\frac{P \cdot \|SSD\|}{FBW} - \frac{T}{DBW} \right) \tag{3.13}$$

Hence, for large enough ranges, MaSM can perfectly overlap the I/O needed to fetch the updates with I/O for the main data if the *time needed to read randomly all of the updates,* $T_U = P \cdot \|SSD\| / FBW$, is less than the *time needed to read sequentially the entire relation,* $T_R = T/DBW$. If this condition does not hold, perfect I/O overlap is still possible provided that the *difference* $T_U - T_R$ *multiplied by the fraction of the range query* is less than *the time needed* *to initiate a main data I/O*, i.e., *the seek time of the HDD storing the main data*. Regarding the second branch of Equation 3.12, i.e., for short ranges that $\|SSD\| \cdot r < N \cdot g$ the condition is:

$$s \geq \frac{N \cdot g \cdot P}{FBW} - \frac{r \cdot T}{DBW} \tag{3.14}$$

Equation 3.14 implies that for short ranges, MaSM can perfectly overlap the I/O needed to fetch the updates with I/O for the main data if the *time needed to read randomly the minimal number of pages per sorted run,* $T_{mU} = \frac{N \cdot g \cdot P}{FBW}$, is less than the *time needed to read sequentially the queried range of the relation,* $T_{rR} = \frac{r \cdot T}{DBW}$, or, if this does not hold, *the difference* $T_{mU} - T_{rR}$ is less than *the seek time of the HDD storing the main data*.

Analysis of MaSM Behavior. We use the analytical model from the previous section to predict the anticipated behavior of the MaSM algorithms for our experimental setup in Section 3.5, as well as, for different setups with varying performance capabilities of the disk subsystem storing the main data and varying capabilities of the SSD storing the updates.

Table 3.2 shows the parameter values considered in our analysis[8]. In particular, for the disk subsystem we use seek time $s = 4.17$ms and disk bandwidth $DBW = 77$MB/s which are either directly measured or taken from device manuals for the experimental machine in the next section. The SSD used shows 75μs read latency and offers 250MB/s read bandwidth. Please note that FBW is not equal to read bandwidth, but it is calculated as the effective bandwidth to read a P-sized page.

In Figure 3.12 we see the I/O time of (i) reading the main data and (ii) reading the updates from the SSD as a function of the query range. The x-axis shows the range in MB for a relation with total size $T = 100$GB. The vertical dashed lines correspond to the ranges we used for our experimentation in Section 3.5.2: 4KB, 1MB, 10MB, 100MB, 1GB, 10GB, 100GB. The blue *Scan only* line shows the I/O time of accessing the main data for the given range, the red lines show the I/O of fetching the updates for different values of α (0.5, 1, 2) and index granularity 64KB. Finally, in the black lines the only change is the granularity which is now 4KB. We observe that for any combination of g and α, MaSM can hide the cost of reading updates if the range is more than 10MB. For $\alpha \leq 1$ and index granularity 4KB, the cost of reading the updates can be hidden for every possible range, even for the smallest possible range of 4KB which models the point query. To detail the comparison between the time to fetch the updates and the time to read the main data we depict the ratio between these two values in Figure 3.13 for index granularity 64KB and in Figure 3.14 for index granularity 4KB.

Figure 3.13 shows that this ratio quickly rises when the range goes below 10MB: fetching the updates may need 5x to 20x times more time than reading the main data for short ranges. In these

---

[8] Some explanations: the total SSD capacity used is 4GB in 64KB pages: $64KB \cdot 65536 = 4GB$. $M = \sqrt{\|SSD\|} = 256$. $N = \alpha M/2$. The index granularity fraction g is 1 for granularity $64KB$ and $1/16$ for granularity $4KB$.

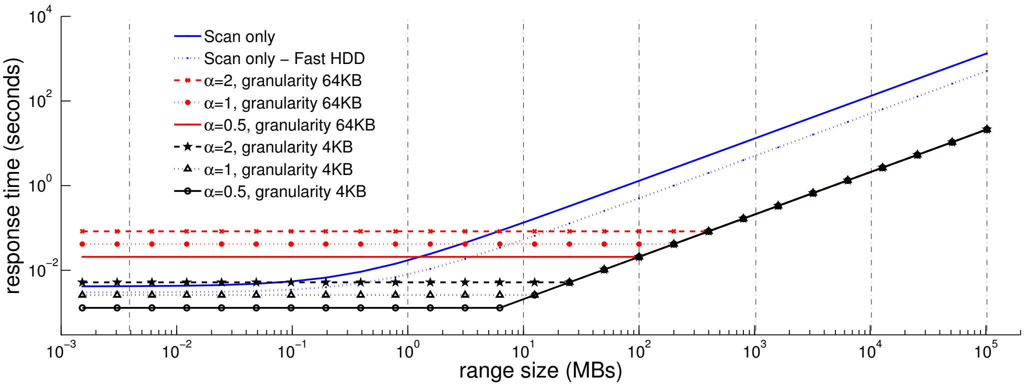


Figure 3.12: Modeling MaSM performance for reading main data and fetching the updates using the characteristics of the storage used in our experimentation.

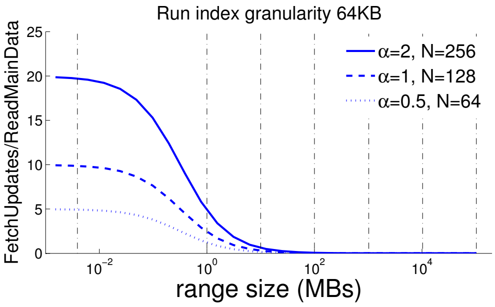


Figure 3.13: The ratio of the time to fetch the updates from SSD to the time to read the main data for index granularity 64KB.

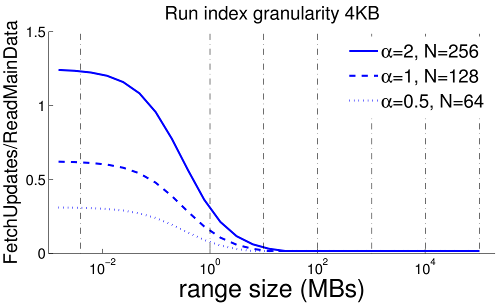


Figure 3.14: The ratio of the time to fetch the updates from SSD to the time to read the main data for index granularity 4KB.

cases MaSM is unable to hide the overhead of reading and merging the updates. On the other hand, Figure 3.14 shows that if we decrease the index granularity from 64KB to 4KB, MaSM with $\alpha = 1$ can read the updates somewhat faster compared with the time needed to read the data of a point query. This behavior is corroborated in Section 3.5.2 when merging updates from the SSD for the point queries leads to small performance penalties. Both Figures 3.13 and 3.14 show that for large ranges starting from 1-10MB the overhead of reading the updates can be easily hidden by the cost of reading the main data.

Next, we calculate the I/O cost of reading the main data from a faster disk, assuming that the seek time is now $s = 3$ms and the disk bandwidth is $DBW = 200$MB/s. The performance of this disk is depicted in Figure 3.12 in the blue dotted line named *Scan only - Fast HDD*. This line shows that if the current system is equipped with a faster disk then the overhead of MaSM grows, particularly for short range queries and one is forced to use less memory to hide the overhead of fetching the updates from the SSD.

The previous analysis of a faster disk opens the question whether MaSM is relevant in a setup involving a read-optimized array of disks, which can deliver the optimal disk performance while

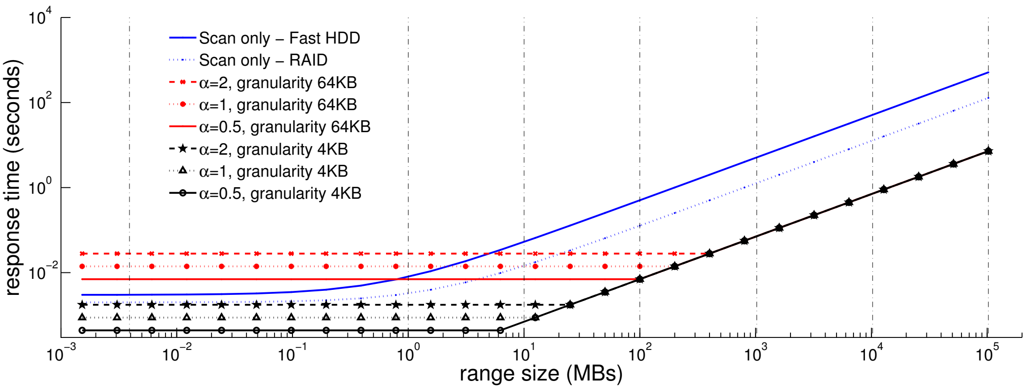


Figure 3.15: Modeling MaSM performance for reading main data and fetching the updates using the characteristics of a high-end storage system.

having larger than before and cost-effective capacity. In fact, in such a setup a single enterprise state-of-the-art flash device suffices to hide the overhead of reading and merging the updates. We argue that if one is equipped with a high-end disk subsystem it is feasible to accompany it with a high-end flash device. Figure 3.15 shows the performance prediction of the previously described *Fast HDD* ($s = 3$ms and $DBW = 200$MB/s) and of a read optimized *RAID* installation ($s = 2$ms and $DBW = 800$MB/s). The flash device is now replaced by a high-end flash device with specifications similar to the ioDrive2 FusionIO card (FusionIO, 2013) (read latency 68μs, read bandwidth 1500MB/s and $FBW = 578$MB/s). The blue dotted line name *Scan only - RAID* shows the I/O time needed for the RAID subsystem to deliver the corresponding range of pages. For the case of a high-end flash device with any possible value of α the MaSM algorithms manage to entirely hide the overhead of fetching the updates when the index granularity is 4KB. Thus, MaSM can be used as an enabler both in a commodity system - as we demonstrate in our experimentation as well - and in a system with high-end storage both for the main data and for the updates.

3.4.5 LSM analysis and comparison with MaSM

In Section 3.2.3 we show that an LSM with $h = 4$ applies minimum writes per update.

Below we give more details for this computation assuming the same storage setup as in our experimentation in Section 3.5.

Table 3.3: Computing the optimal height, h for an LSM tree, using the size ratio between adjacent levels, r, and the I/O write rate normalized by input insertion rate, $(h-0.5)(1+r)$. (The C_0 tree is in memory, and C_1, ..., C_h trees are on SSDs. Suppose memory size allocated to the C_0 tree is 16MB, and the SSD space allocated to the LSM tree is 4GB.)

h	1	2	3	**4**	5	6	7	8	9	10
r	256.0	15.5	6.0	**3.7**	2.8	2.3	2.0	1.8	1.7	1.6
writes/update	128.5	24.8	17.5	**16.5**	17.0	18.1	19.5	21.1	22.7	24.5

LSM optimal height. We compute the optimal height for an LSM tree, which uses 16MB memory and up to 4GB SSD space, as shown in Table 3.3. According to Theorem 3.1 in (O'Neil et al., 1996), in an optimal LSM tree, the ratios between adjacent LSM levels are all equal to a common value r. Therefore, $Size(C_i) = Size(C_0) \cdot r^i$. In Table 3.3, we vary h, the number of LSM levels on SSD, from 1 to 10. For a specific h, we compute r in the second row in the table so that the total size of C_1, ..., C_h (i.e. $C_{tot} = \sum_{i=1}^{h} C_0 \cdot r^i$) for C_0 =16MB, is C_{tot} =4GB. To compute the total I/O writes, we follow the same analysis in the proof of Theorem 3.1 in (O'Neil et al., 1996). Let us consider the rolling merge from C_{i-1} to C_i. Suppose the input insertion rate is R bytes per second. Then the rolling merge must migrate R bytes from C_{i-1} to C_i per second. This entails reading R bytes from C_{i-1}, reading $R * r$ bytes from C_i, and writing $R * (1 + r)$ bytes to C_i per second. Therefore, the LSM tree sees a $R * (1 + r)$ bytes per second I/O write rate at level C_1, ..., C_{h-1}. For the last level, we consider it to grow from 0 to the full size, because when it is close to full, all the update entries in the LSM can be migrated to the main data. On average, the LSM would see $0.5R * (1 + r)$ bytes per second I/O write rate at C_h. Therefore, we can compute the I/O write rate normalized by input insertion rate as $(h - 0.5)(1 + r)$. As shown in Table 3.3, the minimal I/O write rate is achieved when $h = 4$. The optimal LSM tree writes every update entry 16.5 times on SSDs.

MaSM vs LSM. Next, we compare the writes per update for LSM and MaSM when LSM, like MaSM, has two levels. We assume variable memory and SSD space, maintaining, however, the ratio between the two: M pages in memory and M^2 pages on the SSD have as a result a ratio M between C_0 and C_1.

Every time the in-memory buffer fills, its contents are merged with the existing data on the SSD. In the best-case scenario the SSD is initially empty, so, the first time M pages are written on the SSD, the second time M pages are merged with M pages of sorted data (incurring in total $2 \cdot M$ writes), the third time M pages are merged with $2 \cdot M$ pages of sorted data (incurring in total $3 \cdot M$ writes) and so on until the final M^{th} time during which $M \cdot M$ writes take place. In total, during this migration operation, the number of writes is:

$$LSMWrites = M + 2 \cdot M + ... + M \cdot M = 0.5 \cdot M \cdot M \cdot (M + 1) = 0.5 \cdot M^3 + 0.5 \cdot M^2$$

On the other hand, the worst case for MaSM is that every page has been written 2 times on flash, hence leading to total writes equal to $MaSMWrites = 2 \cdot M^2$. Dividing the two quantities we get the ratio between the number of writes for LSM and the number of writes for the worst case of MaSM.

$$\frac{LSMWrites}{MaSMWrites} = \frac{0.5 \cdot M^3 + 0.5 \cdot M^2}{2 \cdot M^2} = 0.25 \cdot M + 0.25$$

The above ratio shows that LSM not only performs more writes than MaSM (since $0.25 \cdot M + 0.25 \geq 1$ for any $M \geq 3$), but additionally it increases the number of writes for larger amount of resources available. On the other hand, as explained in Section 3.4.2, when more memory is available, MaSM manages to decrease the number of writes per update.

3.5 Experimental Evaluation

We perform real-machine experiments to evaluate our proposed MaSM algorithms. We start by describing the experimental setup in Section 3.5.1. Then, we present experimental studies with synthetic data in Section 3.5.2 and perform experiments based on TPC-H traces recorded from a commercial database system in Section 3.5.3.

3.5.1 Experimental Setup

Machine Configuration. We perform all experiments on a Dell Precision 690 workstation equipped with a quad-core Intel Xeon 5345 CPU (2.33GHz, 8MB L2 cache, 1333MHz FSB) and 4GB DRAM running Ubuntu Linux with 2.6.24 kernel. We store the main table data on a dedicated SATA disk (200GB 7200rpm Seagate Barracuda with 77MB/s sequential read and write bandwidth). We cache updates on an Intel X25-E SSD (Intel, 2009) (with 250MB/s sequential read and 170MB/s sequential write bandwidth). All code is compiled with g++ 4.2.4 with "-O2".

Implementation. We implemented a prototype row-store data warehouse, supporting range scans on tables. Tables are implemented as file system files with the slotted page structure. Records are clustered according to the primary key order. A range scan performs 1MB-sized disk I/O reads for high throughput unless the range size is less than 1MB. Incoming updates consist of insertions, deletions, and modifications to attributes in tables. We implemented three algorithms for online updates: (1) In-place updates; (2) IU (Indexed Updates); and (3) MaSM-M. In-place updates perform 4KB-sized read-modify-write I/Os to the main data on disk. The IU implementation caches updates on SSDs and maintains an index to the updates. We model the best performance for IU by keeping its index always in memory in order to avoid random SSD writes to the index. Note that this consumes much more memory than MaSM. Since the SSD has 4KB internal page size, IU uses 4KB-sized SSD I/Os. For MaSM, we experiment with the MaSM-M algorithm, noting that this provides performance lower bounds for any MaSM-αM with $1 \leq \alpha \leq 2$. By default, MaSM-M performs 64KB-sized I/Os to SSDs. Asynchronous I/Os (with libaio) are used to overlap disk and SSD I/Os, and to take advantage of the internal parallelism of the SSD.

Experiments with Synthetic Data (Section 3.5.2). We generate a 100GB table with 100-byte sized records and 4-byte primary keys. The table is initially populated with even-numbered primary keys so that odd-numbered keys can be used to generate insertions. We generate updates randomly uniformly distributed across the entire table, with update types (insertion, deletion, or field modification) selected randomly. By default, we use 4GB of flash-based SSD space for caching updates, thus MaSM-M requires 16MB memory for 64KB SSD effective page size. We also study the impact of varying the SSD space.

TPC-H replay experiments (Section 3.5.3). We ran the TPC-H benchmark with scale factor $SF = 30$ (roughly 30GB database) on a commercial row-store DBMS and obtained the disk traces of the TPC-H queries using the Linux `blktrace` tool. We were able to obtain traces for 20 TPC-H queries except queries 17 and 20, which did not finish in 24 hours. By mapping the I/O addresses in the traces back to the disk ranges storing each TPC-H table, we see that all the 20 TPC-H queries perform (multiple) table range scans. Interestingly, the 4GB memory is large enough to hold the smaller relations in hash joins, therefore hash joins reduce to a single pass algorithm without generating temporary partitions. Note that MaSM aims to minimize memory footprint to preserve such good behaviors.

We replay the TPC-H traces using our prototype data warehouse as follows. We create TPC-H tables with the same sizes as in the commercial database. We replay the query disk traces as the query workload on the real machine. We perform 1MB-sized asynchronous (prefetch) reads for the range scans for high throughput. Then we apply online updates to TPC-H tables using in-place updates and our MaSM-M algorithm. Although TPC-H provides a program to generate batches of updates, each generated update batch deletes records and inserts new records in a very narrow primary key range (i.e. 0.1%) of the `orders` and `lineitem` tables. To model the more general and challenging case, we generate updates to be randomly distributed across the `orders` and `lineitem` tables (which occupy over 80% of the total data size). We make sure that an `orders` record and its associated `lineitem` records are inserted or deleted together. For MaSM, we use 1GB SSD space, 8MB memory, and 64KB sized SSD I/Os.

Measurement Methodology. We use the following steps to minimize OS and device caching effect: (i) opening all the (disk or SSD) files with `O_DIRECT|O_SYNC` flag to get around OS file cache; (ii) disabling the write caches in the disk and the SSD; (iii) reading an irrelevant large file from each disk and each SSD before every experiment so that the device read caches are cleared. In experiments on synthetic data, we randomly select 10 ranges for scans of 100MB or larger, and 100 ranges for smaller ranges. For the larger ranges, we run 5 experiments for every range. For the smaller ranges, we run 5 experiments, each performing all the 100 range scans back-to-back (to reduce the overhead of OS reading file inodes and other metadata). In TPC-H replay experiments, we perform 5 runs for each query. We report the averages of the runs. The standard deviations are within 5% of the averages.

3.5.2 Experiments with Synthetic Data

Comparing All Schemes for Handling Online Updates. Figure 3.16 compares the performance impact of online update schemes on range scan queries on primary key while varying the range size from 4KB (a disk page) to 100GB (the entire table).

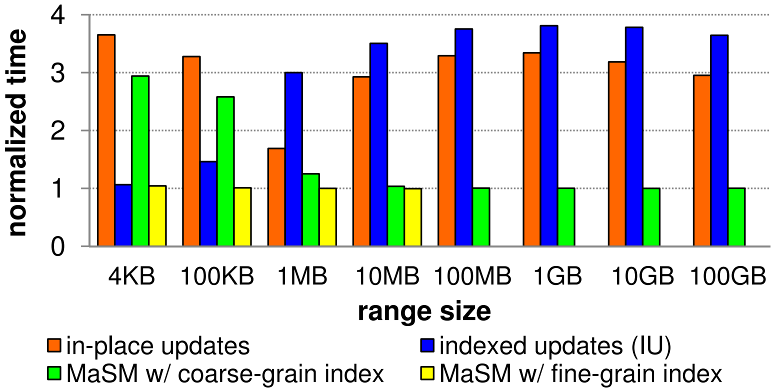


Figure 3.16: Comparing the impact of online update schemes on the response time of primary key range scans (normalized to scans without updates).

For IU and MaSM schemes, the cached updates occupy 50% of the allocated 4GB SSD space (i.e. 2GB), which is the average amount of cached updates expected to be seen in practice. The coarse-grain index records one entry per 64KB cached updates on the SSD, while the fine-grain index records one entry per 4KB cached updates. All the bars are normalized to the execution time of range scans without updates. As shown in Figure 3.16, range scans with in-place updates (the leftmost bars) see 1.7–3.7x slowdowns. This is because the random updates significantly disturb the sequential disk accesses of the range scans. Interestingly, as the range size reduces from 4KB to 1MB, the slowdown increases from 1.7x to 3.7x. We find that elapsed times of pure range scans reduce from 29.8ms to 12.2ms, while elapsed times of range scans with in-place updates reduce from 50.3ms to only 44.7ms. Note that the queries perform a single disk I/O for data size of 1MB or smaller. The single I/O is significantly delayed because of the random in-place updates.

Observing the second bars in Figure 3.16, shows that range scans with IU see 1.1–3.8x slowdowns. This is because IU performs a large number of random I/O reads for retrieving cached updates from the SSD. When the range size is 4KB, the SSD reads in IU can be mostly overlapped with the disk access, leading to quite low overhead for that range size.

MaSM with coarse-grain index incurs little overhead for 100MB to 100GB ranges. This is because the total size of the cached updates (2GB) is only 1/50 of the total data size. Using the coarse-grain run index, MaSM retrieves roughly 1/50 SSD data (cached updates) compared to the disk data read in the range scans. As the sequential read performance of the SSD is higher than that of the disk, MaSM can always overlap the SSD accesses with disk accesses for the large ranges.

For the smaller ranges (4KB to 10MB), MaSM with coarse-grain index incurs up to 2.9x slowdowns. For example, at 4KB ranges, MaSM has to perform 128 SSD reads of 64KB each. This takes about 36ms (mainly bounded by SSD read bandwidth), incurring 2.9x slowdown. On the other hand, MaSM with fine-grain index can narrow the search range down to 4KB SSD pages. Therefore, it performs 128 SSD reads of 4KB each. The Intel X25-E SSD is capable of

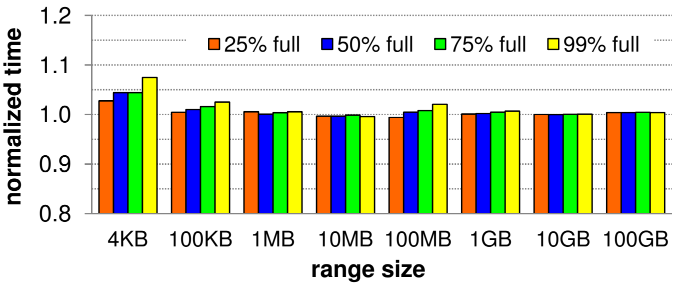


Figure 3.17: MaSM range scans varying updates cached in SSD.

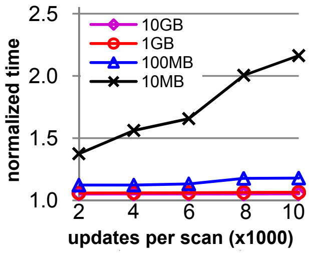


Figure 3.18: Range scan with MaSM varying update speed.

supporting over 35,000 4KB random reads per second. Therefore, the 128 reads can be well overlapped with the 12.2ms 4KB range scan. Overall, MaSM with fine-grain index incurs only 4% overhead even at 4KB ranges.

Comparing Experimental Results with Modeled MaSM's behavior. The above experimental results follow the behavior predicted by the performance model described in Section 3.4.4. MaSM with coarse-grained run indexes results in significant performance degradation for short range scans, while MaSM with fine-grained run indexes hides the merging overhead and shows zero to negligible slowdown for all ranges. According to the model the same pattern is observed when one employs a system with a high-end array of disks for storage of the main data and an enterprise flash-based SSD for buffering the updates.

MaSM Varying Cached Update Size. Figure 3.17 varies both the range size (from 4KB to 100GB) and the cached update size (from 25% full to 99% full) on the SSD. We disable update migration by setting the migration threshold to be 100%. We use MaSM with fine-grain index for 4KB to 10MB ranges, and MaSM with coarse-grain index for 100MB to 100GB ranges. From Figure 3.17, we see that in all cases, MaSM achieves performance comparable to range scans without updates. At 4KB ranges, MaSM incurs only 3%–7% overheads. The results can be viewed from another angle. MaSM with a 25% full 4GB-sized update cache has similar performance to MaSM with a 50% full 2GB-sized update cache. Therefore, Figure 3.17 also represents the performance varying SSD space from 2GB to 8GB with a 50% full update cache.

Varying Update Speed. Figure 3.18 shows the MaSM performance while varying concurrent update speed from 2,000 updates per scan to 10,000 updates per scan. The allocated SSD space is 50% full at the beginning of the experiments. We perform 10 MaSM range scans with concurrent updates. 10,000 updates roughly occupies an 8MB space on the SSD, which is the allocated memory size for MaSM. Therefore, the increase of unsorted updates is between 20% to 100% of M. As shown in the figure, the normalized scan time of MaSM shows a slight upward trend as the update speed increases especially for the smaller ranges. This is because MaSM must process more unsorted data, which cannot be overlapped with the scan operation. The significance of this overhead increases as range size decreases.

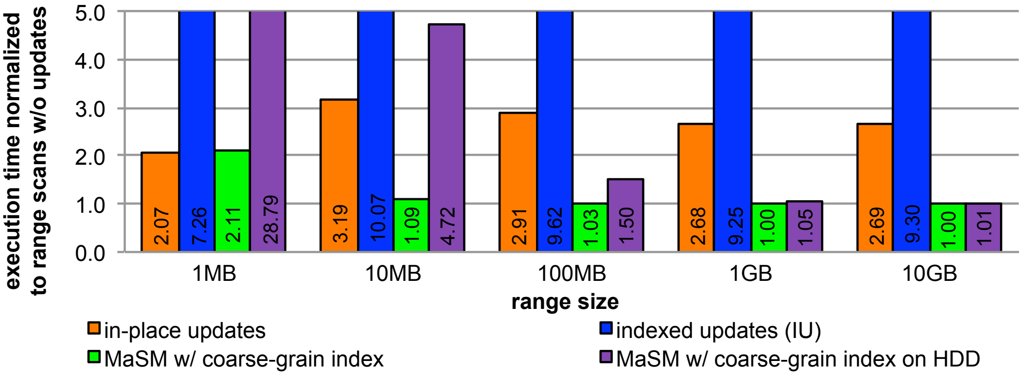


Figure 3.19: Comparing the impact of online update schemes, with MaSM using both SSD and HDD, on the response time of primary key range scans (normalized to scans without updates).

HDD as Update Cache. We experimented using a separate SATA disk (identical to the main disk) as the update cache in MaSM. In this experiment we use a 10GB dataset and the size of the cache is 1GB. Figure 3.19 compares in-place updates, indexed updates (IU), and MaSM with coarse grain index using either SSD or HDD to cache the updates. While the performance trends for the first three configurations are very similar with the respective configurations presented in Figure 3.16, when MaSM uses HDD to cache the updates the overhead for short ranges is impractical. The poor random read performance of the disk-based update cache results in in 28.8x (4.7x) query slowdowns for 1MB (10MB) sized range scans. The overhead of using HDD to cache updates in MaSM diminishes for larger ranges where the cost of reading the cached updates is amortized with the cost to retrieve the main data. This experiment shows that in order to accommodate modern analytical workloads which include both short and long range scans it is essential to cache the updates in a storage medium offering good random read performance, making flash devices a good match.

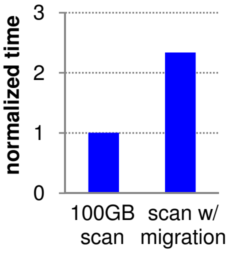


Figure 3.20: MaSM update migration.

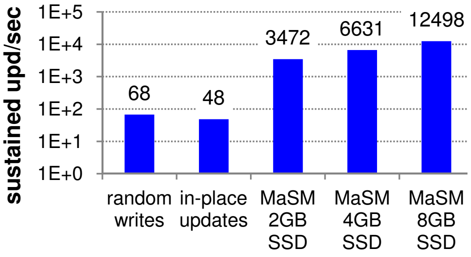


Figure 3.21: Sustained updates per second varying SSD size in MaSM.

Migration Performance. Figure 3.20 shows the performance of migrating 4GB-sized cached updates while performing a table scan. Compared to a pure table scan, the migration performs sequential writes in addition to sequential reads on the disk, leading to 2.3x execution time. The benefits of the MaSM migration scheme are as follows: (i) multiple updates to the same data page are applied together, reducing the total disk I/O operations; (ii) disk-friendly sequential writes rather than random writes are performed; and (iii) it updates main data in place. Finally, note that MaSM incurs its migration overhead orders of magnitude less frequently than prior approaches—recall Figure 3.1.

Sustained Update Rate. Figure 3.21 reports the sustained update throughput of in-place updates, and three MaSM schemes with different SSD space. For in-place updates, we depict the best update rate by performing only updates, without concurrent queries. For MaSM, we continuously perform table scans. The updates are sent as fast as possible so that every table scan incurs the migration of updates back to the disk. We set the migration threshold to be 50% so that in steady state, a table scan with migration is migrating updates in 50% of the SSD while the other 50% of the SSD is holding incoming updates. Figure 3.21 also shows the disk random write performance. We see that (i) compared to in-place updates, which perform random disk I/Os, MaSM schemes achieve orders of magnitude higher sustained update rates; and (ii) as expected, doubling the SSD space roughly doubles the sustained update rate.

General Transactions with Read-Modify-Writes. Given the low overhead of MaSM even at 4KB ranges, we argue that MaSM can achieve good performance for general transactions. With MaSM, the reads in transactions achieve similar performance as if there were no online updates. On the other hand, writes are appended to the in-memory buffer, resulting in low overhead.

Varying CPU Cost of Query Processing. Complex queries may perform significant amount of in-memory processing after retrieving records from range scans. We study how MaSM behaves when the range scan becomes CPU bound. In Figure 3.22, we model query complexity by injecting CPU overhead. For every 1000 retrieved records, we inject a busy loop that takes 0.5ms, 1.0ms, 1.5ms, 2.0ms, or 2.5ms to execute. In other words, we inject 0.5us to 2.5us CPU cost per record. As shown in Figure 3.22, the performance is almost flat until the 1.5us point, indicating that the range scan is I/O bound. From 1.5us to 2.5us, the execution time grows roughly linearly, indicating that the range scan is CPU bound. Most importantly, we see that range scans with MaSM have indistinguishable performance compared with pure range scans for all cases. The CPU overhead for merging the cached updates with main data is insignificant compared to (i) asynchronous I/Os when the query is I/O bound and (ii) in-memory query overhead when the query is CPU bound. The sorted runs containing the cached updates are merged using heap-merge, hence, having complexity logarithmic to the number of sorted-runs, $O(log(\#runs))$.

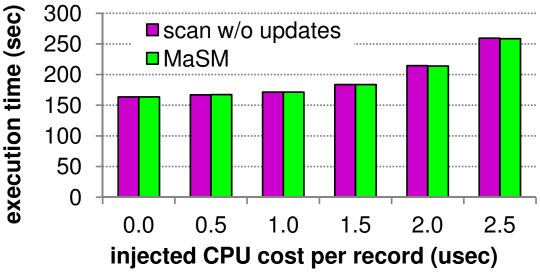


Figure 3.22: Range scan and MaSM performance while emulating CPU cost of query processing (10GB ranges).

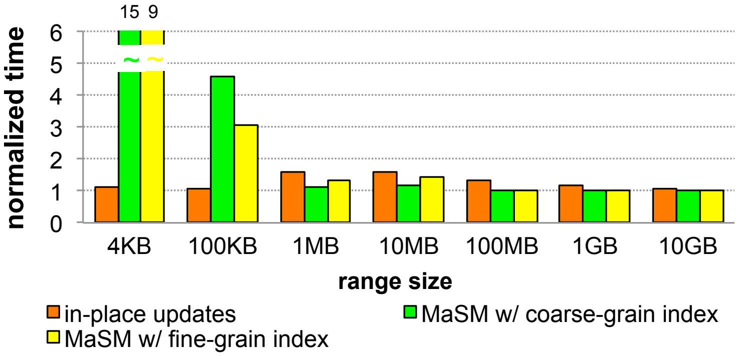


Figure 3.23: Comparing in-place updates with MaSM performance (using both coarse-grain and fain-grain run indexes) when both data and updates are stored in memory (10GB table size).

Merging Overhead. We further investigate the merging overhead by running MaSM while storing both main data and updates on a ramdisk. This is a pure CPU-bound execution, hence, any slowdowns are due to the merging overhead of MaSM. To run this experiment we use a server with more main memory: a Red-Hat Linux server with 2.6.32 64-bit kernel, equipped with 2 6-Core 2.67GHz Intel Xeon CPU X5650 and 48GB of main memory. For this experiment, we generate a 10GB table with 100-byte sized records and 4-byte primary keys. Similarly to the previous experiments, the table is initially populated with even-numbered primary keys so that odd-numbered keys can be used to generate insertions. We generate updates randomly uniformly distributed across the entire table, with update types (insertion, deletion, or field modification) selected randomly. We use 1GB of in-memory space for caching updates, thus MaSM-M requires 8MB memory for 64KB effective page size. Figure 3.23 compares the performance of range scans using in-place updates and MaSM (both with coarse-grain and fine-grain indexes), varying the range size from 4KB (a disk page) to 10GB (the entire table). Similarly to previous experiments, coarse-grain indexes record one entry per 64KB cached updates, and fine-grain index records one entry per 4KB cached updates. When comparing coarse-grain with fine-grain indexes, note that coarse-grain indexes require to read bigger chunks of the sorted runs but do less comparisons, while fine-grain indexes read more smaller chunks and, as a result, need to do more comparisons. For both MaSM schemes, the cached updates occupy about 50% of the allocated 1GB space, which is the average amount of cached updates expected to be seen in practice. All the bars are normalized to the execution time of range scans without updates.

As shown in Figure 3.23, contrary to the I/O bound experiments, range scans with in-place updates see small performance penalty, varying between 7% and 58%. On the other hand, both MaSM schemes show significant performance penalty for 4KB and 100KB ranges (varying from 3x to 15x). For 1MB and 10MB ranges MaSM sees small performance degradation (12% to 42%) and for 100MB, 1GB and 10GB ranges, MaSM sees no performance penalty. Similarly to the I/O-bound experiments, in CPU-bound experiments the MaSM overhead is hidden when a large portion of the main file is requested by the range scan. The CPU overhead of MaSM consists of two parts. The first part is the initial operation to search for the starting update record for a given range. The second part is the CPU overhead for merging update records with main data. When the range size is large, the second part dominates. From the figure, it is clear that this overhead is negligible. That means the overhead for merging updates with main data is very small. When the

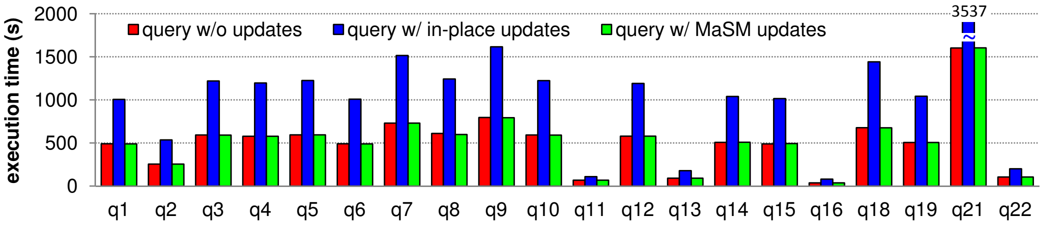


Figure 3.24: Replaying I/O traces of TPC-H queries on a real machine with online updates.

range size is small, the first part becomes important, which is the reason for the large overhead for the 4KB and 100KB cases.

Overall, as expected, the merging overhead of MaSM for very short range scans cannot be amortized in the in-memory case. On the other hand, for range scans that are more than 0.01% of the file, MaSM offers better performance than in-place updates even when data are stored in the main memory. An observation that is entirely hidden in the I/O-bound experiments is that for intermediate ranges (1MB and 10MB) the comparison overhead of the fine-grain run indexes exceeds the benefits of the reduced amount of fetched data compared with coarse-grain run indexes, leading, nevertheless to smaller slowdowns when compared with range scans with in-place updates.

3.5.3 TPC-H Replay Experiments

Figure 3.24 shows the execution times of the TPC-H replay experiments (in seconds). The left bar is the query execution time without updates; the middle bar is the query execution time with concurrent in-place updates; the right bar is the query execution time with online updates using MaSM. For the MaSM algorithm, the SSD space is 50% full at the start of the query. MaSM divides the SSD space to maintain cached updates on a per table basis in the TPC-H experiments).

From Figure 3.24, we see that in-place updates incur 1.6–2.2x slowdowns. In contrast, compared to pure queries without updates, MaSM achieves very similar performance (with up to 1% difference), providing fresh data with little I/O overhead. Note that the queries typically consist of multiple (concurrent) range scan operations on multiple tables. Therefore, the results also show that MaSM can handle multiple concurrent range scans well.

3.6 Discussion

3.6.1 Shared-Nothing Architectures

Large analytical data warehouses often employ a shared-nothing architecture for achieving scalable performance (Becla and Lim, 2008). The system consists of multiple machine nodes with local storage connected by a local area network. The main data are distributed across multiple machine nodes by using hash partitioning or range partitioning. Incoming updates are mapped and sent to individual machine nodes, and data analysis queries often are executed in parallel on many machine nodes. Because updates and queries are eventually decomposed into operations on individual machine nodes, MaSM algorithms can be applied on a per-machine-node basis. Note that recent data center discussions show that it is reasonable to enhance every machine node with SSDs (Barroso, 2010).

MaSM addresses the challenge to incorporate the most recent updates in the main data residing in a node with negligible overhead. A similar concept can be implemented for shared-nothing architectures where scale-out is the primary goal. In a scale-out setup multiple nodes host the data, and the updates are stored either in a single or in multiple nodes. There are three possible approaches to exploit MaSM in scale-out setup. First, MaSM can enhance the scan operator of each node transparently to the rest of the system both in the single node, and in the API between different nodes. Second, MaSM can be designed in a hierarchical fashion enhancing each node with SSDs to cache updates. In order to answer a query a MaSM operator in the inter-node level will be instantiated which will merge updates and data from every node involved. In this case, each node holds the updates corresponding to the data residing in the node which could potentially lead to unbalanced load. In order to amend this drawback, the third proposal, caches all updates in all nodes and feeds to the inter-node level MaSM operator only tuples after they have been merged with the most recent updates.

3.6.2 MaSM for deeper memory hierarchies and new storage technologies

MaSM is designed to exploit the increased depth of memory hierarchies offered by flash devices. Typically, flash-based SSDs and high-end flash devices are perceived today either as secondary storage, or as a level of the memory hierarchy residing between main memory and secondary storage. Flash devices, however, are fundamentally different than the other levels since they have a read/write performance asymmetry and endurance limitations (discussed in Section 2.1). Hence, in several approaches that flash is used as a caching layer, including MaSM, flash has a specialized role tailored for its characteristics and respecting its limitations.

The memory hierarchy is constantly evolving and more technologies are expected to augment it, potentially by adding more levels. Technologies like PCM, ferroelectric RAM, magnetic RAM, and Memristor may achieve a viable memory product which can reside performance-wise and price-wise between main memory and flash devices, reducing, as well, the effect of limited endurance that flash devices started hurting from. MaSM can naturally exploit new technologies by either replacing flash with its successor, or by exploiting a deeper hierarchy. Assuming, for example, that a new level of PCM non-volatile main memory will be added in the system, MaSM can be altered as follows. All incoming updates can be safely kept "in-memory" without the need of an external logging mechanism, while the sorted runs will be written out on the flash device (which has limited endurance). Taking this idea a step further, in the generic MaSM-αM algorithm, incoming updates and first-pass sorted runs will be written on non-volatile main memory and second-pass sorted runs on flash. Additionally, some second-pass sorted runs can be kept in the newly introduced level to keep updates that are often used when answering queries in a faster level.

3.6.3 Can SSDs Enhance Performance and Be Cost Efficient?

SSDs have been studied as potential alternatives or enhancements to HDDs in many recent database studies (Bouganim et al., 2009; Canim et al., 2009, 2010b; Chen, 2009; Chen et al., 2010; Koltsidas and Viglas, 2008; Lee et al., 2008; Ozmen et al., 2010; Stoica et al., 2009; Tsirogiannis et al., 2009). In this chapter, we exploit SSDs to cache differential updates. Here, we perform a back-of-the-envelope computation to quantify the cost benefits of using SSDs.

Assuming we have a budget B for storage, we can either spend it on N disks or spend $X\%$ for disks and the rest for SSD. If $N \cdot X\%$ disks can provide the desired capacity then the performance gain of running a table scan when using SSD can be computed as follows. A table scan is linearly faster with the number of disks, while random updates slows performance down by a constant factor F_{HDD} which ranges between 1.5 and 4, as shown in Figure 3.3. When executing a table scan with concurrent random updates the performance gain is N/F_{HDD}. If we buy SSD with $100\% - X\%$ of our budget, on the one hand we decrease the performance gain due to disks to $N \cdot X\%$; on the other hand, the update cost is slowed down with a different constant factor F_{SSD}, which is close to 1 in our experiments. The performance gain now is $(N \cdot X\%)/F_{SSD}$. Therefore, the relative performance gain of using SSD is computed as follows:

$$RelativeGain = \frac{N \cdot X\%}{F_{SSD}} \cdot \frac{F_{HDD}}{N} = \frac{F_{HDD} \cdot X\%}{F_{SSD}}$$

We can select the budget used for SSD to be low enough to ensure the desired overall capacity and high enough to lead to important performance gain. For example, if $X\% = 90\%$ of the budget is used for the disk storage, $RelativeGain$ of using SSD as in our proposal is between 1.35 and 3.6.

3.6.4 General support of MaSM

Secondary Index. We discuss how to support index scans in MaSM. Given a secondary index on Y and a range $[Y_{begin}, Y_{end}]$, an index scan is often served in two steps in a database. Similar to index-to-index navigation (Graefe, 2010), in the first step, the secondary index is searched to retrieve all the record pointers within the range. In the second step, the record pointers are used to retrieve the records. Thus, any query retrieving a range of k records using a secondary index can be transformed to k point queries on the primary index. We show that point queries on the primary index can be handled efficiently, so, a query using a secondary index will be handled correctly and efficiently. An optimization for disk performance is to sort the record pointers according to the physical storage order of the records between the two steps. In this case, instead of k point queries, a query on a secondary index will be transformed to a number of range queries (less than or equal to k), having, as a result, better I/O performance.

For every retrieved record, MaSM can use the key (primary key or RID) of the record to look up corresponding cached updates and then merge them. However, we must deal with the special case where Y is modified in an incoming update: We build a *secondary update index* for all the update records that contain any Y value, comprised of a read-only index on every materialized sorted run and an in-memory index on the unsorted updates. The index scan searches this secondary update index to find any update records that fall into the desired range $[Y_{begin}, Y_{end}]$. In this way, MaSM can provide functionally correct support for secondary indexes.

Multiple Sort Orders. Heavily optimized for read accesses, column-store data warehouses can maintain multiple copies of the data in different sort orders (a.k.a. projections) (Stonebraker et al., 2005; Lamb et al., 2012). For example, in addition to a prevailing sort order of a table, one can optimize a specific query by storing the columns in an order that is most performance friendly for the query. However, multiple sort orders present a challenge for differential updates; prior work does not handle this case (Héman et al., 2010).

One way to support multiple sort orders would be to treat columns with different sort orders as different tables, and to build different update caches for them. This approach would require that every update must contain the sort keys for all the sort orders so that the RIDs for individual sort orders could be obtained.

Alternatively, we could treat sort orders as secondary indexes. Suppose a copy of column X is stored in an order O_X different from the prevailing RID order. In this copy, we store the RID along with every X value so that when a query performs a range scan on this copy of X, we can use the RIDs to look up the cached updates. Note that adding RIDs to the copy of X reduces compression effectiveness, because the RIDs may be quite randomly ordered. Essentially, X with RID column looks like a secondary index, and can be supported similarly.

Materialized Views. Materialized views can speed up the processing of well-known query types. A recent study proposed lazy maintenance of materialized views in order to remove view maintenance from the critical path of incoming update handling (Zhou et al., 2007). Unlike eager view maintenance where the update statement or the update transaction eagerly maintains any affected views, lazy maintenance postpones the view maintenance until the data warehouse has free cycles or a query references the view. It is straightforward to extend differential update schemes to support lazy view maintenance, by treating the view maintenance operations as normal queries.

3.6.5 Applying MaSM to Cloud Data Management

Key-Value stores are becoming a popular cloud data storage platform because of its scalability, availability and simplicity. Many Key-Value store implementations (e.g., Bigtable, HBase, and Cassandra) follow the LSM principle to handle updates. In Section 3.1.2, we have discussed the difference between these schemes and MaSM. In the past several years, SSD capacity has been increasing dramatically, and its price per capacity ratio is reducing steadily. This makes SSDs increasingly affordable; SSDs have already been considered as standard equipment for cloud machines (Barroso, 2010). When a Key-Value store node is equipped with an SSD, the presented MaSM algorithm can be applied to improve the performance, reduce memory footprint, and reduce SSD wear in Key-Value stores.

3.7 Conclusion

Efficient analytical processing in applications that require data freshness is challenging. The conventional approach of performing random updates in place degrades query performance significantly, because random accesses disturb the sequential disk access patterns of the typical analysis query. Recent studies follow the differential update approach, by caching updates separate from the main data and combining cached updates on-the-fly in query processing. However, these proposals all require large in-memory buffers or suffer from high update migration overheads.

In this chapter, we proposed to judiciously use solid-state storage to cache differential updates. Our work is based on the principle of using SSD as a performance booster for databases stored primarily on magnetic disks, since for the foreseeable future, magnetic disks offer cheaper capacity (higher GB/$), while SSDs offer better but more expensive read performance (higher IOPS/$). We presented a high-level framework for SSD-based differential update approaches, and identified five design goals. We presented an efficient algorithm, MaSM, that achieves low query overhead, small memory footprint, no random SSD writes, few SSD writes, efficient in-place migration, and correct ACID support. Experimental results using a prototype implementation showed that, using MaSM, query response times remain nearly unaffected even if updates are running at the same time. Moreover, update throughput using the MaSM algorithms is several orders of magnitude higher than in-place updates and, in fact, tunable by selecting different SSD buffer capacity.

4 Enhancing Data Analytics with Work Sharing[1]

4.1 Introduction

Data warehouses (DW) are database systems specialized for servicing on-line analytical processing (OLAP) workloads. OLAP workloads consist mostly of ad-hoc, long running, scan-heavy queries over relatively static data (new data are periodically loaded). Today, in the era of data deluge (Kersten et al., 2011), organizations collect massive data for analysis. The increase of processing power and the growing applications' needs have led to increased requirements for both throughput and latency of analytical queries over ever-growing datasets.

General-purpose DW, however, cannot easily handle analytical workloads over big data with high concurrency (Becla and Lim, 2008; Candea et al., 2009). A limiting factor is their typical ***query-centric*** model: DW optimize and execute each query independently. Concurrent queries, however, often exhibit overlapping data accesses or computations. The query-centric model misses the opportunities of sharing work and data among them and results in performance degradation due to the contention of concurrent queries for I/O, CPU and RAM resources.

4.1.1 Methodologies for sharing data and work

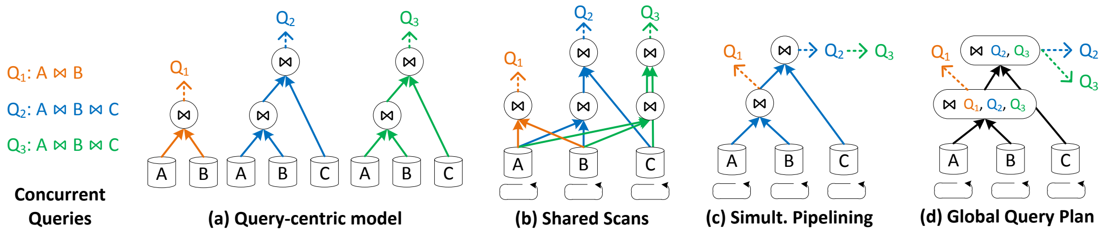
A variety of ideas have been proposed to exploit sharing, including diverse buffer pool management techniques, materialized views, caching and multi-query optimization (see Section 2.3). More recently, query engines started sharing data at the I/O layer using ***shared scans*** (a technique with variants also known as circular scans, cooperative scans or clock scan) (Colby et al., 1998; Cook, 2001; Morri, 2002; Qiao et al., 2008; Unterbrunner et al., 2009; Zukowski et al., 2007).

In this chapter, we evaluate work sharing techniques at the level of the execution engine, where we distinguish two predominant methodologies: (a) Simultaneous pipelining (SP) (Harizopoulos et al., 2005), and (b) Global query plans (GQP) (Arumugam et al., 2010; Candea et al., 2009,

---

[1] The material of this chapter has been the basis for the VLDB 2013 paper "Sharing Data and Work Across Concurrent Analytical Queries" (Psaroudakis et al., 2013).

2011; Giannikis et al., 2012) with shared operators.



(a) Query-centric model (b) Shared Scans (c) Simult. Pipelining (d) Global Query Plan

Figure 4.1: Evaluation of three concurrent queries using (a) a query-centric model, (b) shared scans, (c) SP, and (d) a GQP.

Simultaneous pipelining (SP) is introduced in QPipe (Harizopoulos et al., 2005), an operator-centric execution engine, where each relational operator is encapsulated into a self-contained module called a ***stage***. Each stage detects common sub-plans among concurrent queries, evaluates only one of them and pipelines the results to all common sub-plans simultaneously when possible (see Sections 2.3.2 and 2.3.3).

Global query plans (GQP) with shared operators are introduced in the CJOIN operator (Candea et al., 2009, 2011). A single shared operator is able to evaluate multiple concurrent queries. CJOIN uses a GQP, consisting of shared hash-join operators that evaluate the joins of multiple concurrent queries simultaneously. More recent research prototypes extend the logic to additional operators and to more general cases (Arumugam et al., 2010; Giannikis et al., 2012) (see Sections 2.3.4 and 2.3.5).

Figure 4.1 illustrates how a query-centric model, shared scans, SP, and a GQP operate through a simple example of three concurrent queries which perform natural joins without selection predicates and are submitted at the same time. The last two queries have a common plan, which subsumes the plan of the first. We note that shared scans are typically used with both SP and GQP.

4.1.2 Integrating Simultaneous Pipelining and Global Query Plans

In order to perform our analysis and experimental evaluation of SP vs. GQP, we integrate the original research prototypes that introduced them into one system: We integrate the CJOIN operator as an additional stage of the QPipe execution engine (see Section 4.2). Thus, we can dynamically decide whether to evaluate multiple concurrent queries with the standard query-centric relational operators of QPipe, with or without SP, or the GQP offered by CJOIN.

Furthermore, this integration allows us to combine the two sharing techniques, showing that they are in fact orthogonal. As shown in Figure 4.1d, the GQP misses the opportunity of sharing common sub-plans, and redundantly evaluates both $Q2$ and $Q3$. SP can be applied to shared operators to complement a GQP with the additional capability of sharing common sub-plans (see Section 4.2).

4.1.3 Optimizing Simultaneous Pipelining

For the specific case of SP, it is shown in the literature (Johnson et al., 2007; Qiao et al., 2008) that if it entails a serialization point, then enforcing aggressive sharing does not always improve performance. In cases of low concurrency and enough available resources, it is shown that the system should first parallelize with a query-centric model before sharing.

To calculate the turning point where sharing becomes beneficial, a prediction model is proposed (Johnson et al., 2007) for determining at run-time whether SP is beneficial. In this chapter, however, we show that the serialization point is due to the push-based communication employed by SP (Harizopoulos et al., 2005; Johnson et al., 2007). We show that pull-based communication can drastically minimize the impact of the serialization point, and is better suited for sharing common results on multi-core machines.

We introduce ***Shared Pages Lists*** (SPL), a pull-based sharing approach that eliminates the serialization point caused by push-based sharing during SP. SPL are data structures that store intermediate results - of relational operators - and allow for a single producer and multiple consumers. SPL make sharing with SP always beneficial and reduce response times by 82%-86% in cases of high concurrency, compared to the original push-based SP design and implementation (Harizopoulos et al., 2005; Johnson et al., 2007) (see Section 4.3).

4.1.4 Simultaneous Pipelining vs. Global Query Plans

Having optimized SP, and having integrated the CJOIN operator in the QPipe execution engine, we proceed to perform an extensive analysis and experimental evaluation of SP vs. GQP (see Section 4.5). Our work answers two fundamental questions: *when* and *how* an execution engine should share in order to improve performance of typical analytical workloads.

Sharing in the execution engine. We identify a performance trade-off between using a query-centric model and sharing. For a high number of concurrent queries, the execution engine should share, as it reduces contention for resources and improves performance in comparison to a query-centric model. For low concurrency, however, sharing is not always beneficial.

With respect to SP, we corroborate previous related work (Johnson et al., 2007; Qiao et al., 2008), that if SP entails a serialization point, then enforcing aggressive sharing does not always improve performance in cases of low concurrency. Our newly optimized SP with SPL, however, eliminates the serialization point, making SP beneficial in cases of both low and high concurrency.

With respect to GQP, we corroborate previous work (Arumugam et al., 2010; Candea et al., 2009, 2011; Giannikis et al., 2012) that shared operators are efficient in reducing contention for resources and in improving performance for high concurrency (see Section 4.5.1). The design of a shared operator, however, inherently increases book-keeping in comparison to the typical operators of a query-centric model. Thus, for low concurrency, we show that shared operators result in worse performance than the traditional query-centric operators (see Section 4.5.2).

Moreover, we show that SP can be applied to shared operators of a GQP, in order to get the best out of the two worlds. SP can reduce the response time of a GQP by 20%-48% for workloads with common sub-plans (see Section 4.5.3).

Sharing in the I/O layer. Though our work primarily studies sharing inside the execution engine, our experimental results also corroborate previous work relating to shared scans. The simple case of a circular scan per table is able to improve performance of typical analytical workloads both in cases of low and high concurrency. In highly concurrent cases, response times are reduced by 80%-97% in comparison to independent table scans (see Section 4.5.1).

Rules of thumb. Putting all our observations together, we deduce a few rules of thumb for sharing, presented in Table 4.1. Our rules of thumb apply for the case of typical OLAP workloads involving ad-hoc, long running, scan-heavy queries over relatively static data.

When	*How* to share in the	
	Execution Engine	I/O Layer
Low concurrency	Query-centric operators + SP	Shared Scans
High concurrency	GQP (shared operators) + SP	

Table 4.1: Rules of thumb for when and how to share data and work across typical concurrent analytical queries in DW.

4.1.5 Contributions

In this chapter, we perform an experimental analysis of two work sharing methodologies, (a) Simultaneous Pipelining (SP), and (b) Global Query Plans (GQP), based on the original research prototypes that introduced them. Our analysis answers two fundamental questions: *when* and *how* an execution engine should employ sharing in order to improve performance of typical analytical workloads. More details on categorizing different sharing techniques for relational databases, as well as for SP and GQP, the two state-of-the-art sharing methodologies for the execution engine can be found in Section 2.3. Next, our work makes the following main contributions:

- **Integration of SP and GQP:** We show that SP and GQP are orthogonal, and can be combined to take the best of the two worlds (Section 4.2). In our experiments, we show that SP can further improve the performance of a GQP by 20%-48% for workloads that expose common sub-plans.
- **Pull-based SP:** We introduce Shared Pages Lists (SPL), a pull-based approach for SP that eliminates the sharing overhead of push-based SP. Pull-based SP is better suited for multi-core machines than push-based SP, is beneficial for cases of both low and high concurrency, and further reduces response times by 82%-86% for high concurrency (Section 4.3).

- **Evaluation of SP vs. GQP:** We analyze the trade-offs of SP, GQP and their combination, and we detail through an extensive sensitivity analysis when each one is beneficial (Section 4.5). We show that query-centric operators combined with SP result in better performance for cases of low concurrency, while GQP with shared operators enhanced by SP are better suited for cases of high concurrency.

4.1.6 Outline

This chapter is organized as follows. Section 4.2 describes our implementation for integrating SP and GQP. Section 4.3 presents shared pages lists, our pull-based solution for sharing common results during SP. Section 4.5 includes our experimental evaluation. Section 4.6 includes a short discussion. We present our conclusions in Section 4.7.

4.2 Integrating SP and GQP

By integrating SP and GQP, we can exploit the advantages of both forms of sharing: Shared operators in a GQP are efficient in handling a large number of concurrent similar queries, while SP exploits common sub-plans in the query mix.

In Section 4.2.1, we describe how SP can conceptually improve performance of shared operators in a GQP in the presence of common sub-plans. We continue in Sections 4.2.2 and 4.2.3 to describe our implementation based on CJOIN and QPipe.

4.2.1 Benefits of applying SP to GQP

In this section, we mention several conceptual examples, showing the advantages of applying SP to shared operators in a GQP. These observations apply to general GQP, and are thus applicable to all the research prototypes we mention in Section 2.3.4.

Identical queries. If a new query is completely identical with an on-going query, SP takes care to re-use the final results of the on-going query for the new query. If we assume that the top-most operators in a query plan have a full step WoP (e.g. when final results are buffered and given wholly to the client instead of being pipelined), the new query does not need to participate at all in the GQP, independent of its time of arrival during the on-going query's evaluation. This is the case where the integration of SP and GQP offers the maximum performance benefits. Additionally, admission costs are completely avoided, the tuples' bitmaps do not need to be extended to accommodate the new query (translating to fewer bit-wise operations), and the latency of the new query is decreased to the latency of the remaining part of the on-going query.

Shared selections. If a new query has the same selection predicate as an on-going query, SP allows to avoid the redundant evaluation of the same selection predicate from the moment the new query arrives until the end of evaluation of the on-going query (a selection operator has a linear WoP). For each tuple, SP copies the resulting bit of the shared selection operator for the on-going query, to the position in the tuple's bitmap that corresponds to the new query.

Shared joins. If a new query has a common sub-plan with an on-going query under a shared join operator, and arrives within the step WoP, SP can avoid extending tuples' bitmaps with one more bit for the new query for the sub-plan. The join still needs to be evaluated, but the number of bit-wise AND operations is decreased.

Shared aggregations. If a new query has a common sub-plan with an on-going query under a shared aggregation operator, and arrives within the step WoP, SP avoids calculating a redundant sum. It copies the final result from the on-going query.

Admission costs. For every new query submitted to a GQP, an admission phase is required that possibly re-adjusts the GQP to accommodate it. In case of common sub-plans, SP can avoid part of the admission costs. The cost depends on the implementation.

For CJOIN (Candea et al., 2009, 2011), the admission cost of a new query includes (a) scanning all involved dimension tables, (b) evaluating its selection predicates, (c) extending the bitmaps attached to tuples, (d) increasing the size of hash tables of the shared hash-joins to accommodate newly selected dimension tuples (if needed), and (e) stalling the pipeline to re-adjust filters (Candea et al., 2009, 2011). In case of identical queries, SP can avoid these costs completely. In case of common sub-plans, SP can avoid parts of these costs, such as avoiding scanning dimension tables for which selection predicates are identical.

For DataPath (Arumugam et al., 2010), SP can decrease the optimization time of the GQP if it assumes that the common sub-plan of a new query can use the same part of the current GQP as the on-going query. For SharedDB (Giannikis et al., 2012), SP can help start a new query before the next batch at any operator, if it has a common sub-plan with an on-going query and has arrived within the corresponding WoP of the operator.

4.2.2 CJOIN as a QPipe stage

We integrate the original CJOIN operator into the QPipe execution engine as a new stage, using Shore-MT (Johnson et al., 2007) as the underlying storage manager. In Figure 4.2, we depict the new stage that encapsulates the CJOIN pipeline.

The CJOIN stage accepts incoming QPipe packets that contain the necessary information to formulate a star query: (a) the projections for the fact table and the dimension tables to be joined, and (b) the selection predicates. The CJOIN operator does not support selection predicates for the fact table (Candea et al., 2009, 2011), as these would slow the preprocessor significantly. If the preprocessor evaluates the fact table selection predicates, it results to a slower pipeline, which defeats the purpose of potentially flowing fewer fact tuples in the pipeline. Fact table predicates

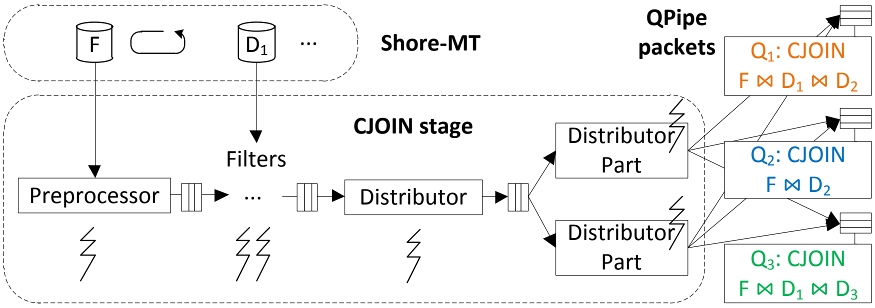


Figure 4.2: Integration scheme of CJOIN as a QPipe stage.

are evaluated on the output tuples of CJOIN.

To improve admission costs we use batching, following the original CJOIN proposal (Candea et al., 2011). In one pause of the pipeline, the admission phase adapts filters for all queries in the batch. While the pipeline is paused, additional new queries form a new batch to be admitted after the current batch is finished.

With respect to threads, there is a number of threads assigned to filters (we assume the single stage model (Candea et al., 2009, 2011)), each one taking a fact tuple from the preprocessor, passing it through the filters and giving it to the distributor. The original CJOIN uses a single-threaded distributor which slows the pipeline significantly. To address this bottleneck, we augment the distributor with several *distributor parts*. Every distributor part takes a tuple from the distributor, examines its bitmap, and determines relevant CJOIN packets. For each relevant packet, it performs the projection of the star query and forwards the tuple to the output buffer of the packet.

CJOIN supports only shared hash-joins. Consequent operators in a query plan, e.g. aggregations or sorts, are query-centric. Nevertheless, our evaluation gives us insight on the general behavior of shared operators in a GQP, as joins typically comprise the most expensive part of a star query.

4.2.3 SP for the CJOIN stage

We enable SP for the CJOIN stage with a step WoP. Evaluating the identical queries Q_2 and Q_3 of Figure 4.1d employing SP, requires only one packet entering the CJOIN pipeline. The second satellite packet re-uses the results.

CJOIN is itself an operator, and we integrate it as a new stage in QPipe. As with any other QPipe stage, SP is applied on the overall CJOIN stage. Conceptually, our implementation applies SP for the whole series of shared hash-joins in the GQP. Our analysis, however, gives insight on the benefits of applying SP to fine-grained shared hash-joins as well. This is due to the fact that a redundant CJOIN packet involves all redundant costs we mentioned in Section 4.2.1 for admission, shared selections operators and shared hash-joins. Our experiments show that the cost of a redundant CJOIN packet is significant, and SP decreases it considerably.

4.3 Shared Pages Lists for SP

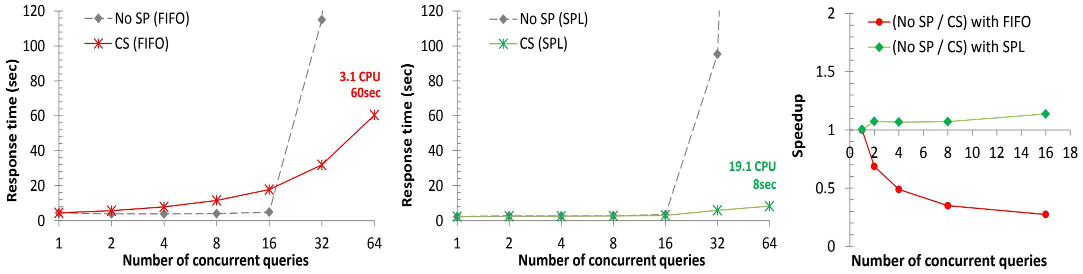


Figure 4.3: Evaluating multiple identical TPC-H Q1 queries (a) with a push-only model during SP (FIFO), and (b) with a pull-based model during SP (SPL). In (c), we show the corresponding speedups of the two methods of SP, over not sharing, for low concurrency.

In this section, we present design and implementation issues of sharing using SP, and how to address them. Contrary to intuition, it is shown in the literature that work sharing is not always beneficial: if there is a serialization point during SP, then sharing common results aggressively can lead to worse performance, compared to a query-centric model that implicitly exploits parallelism (Johnson et al., 2007; Qiao et al., 2008). When the producer (host packet) forwards results to consumers (satellite packets), it is in the critical path of the evaluation of the remaining nodes of the query plans of all involved queries. Forwarding results can cause a significant serialization point. In this case, the DBMS should first attempt to exploit available resources and parallelize as much as possible with a query-centric model, before sharing. A prediction model is proposed (Johnson et al., 2007) for determining at run-time whether sharing is beneficial. In this section, however, we show that SP is possible without a serialization point, thus rendering SP always beneficial.

The serialization point is caused by strictly employing push-only communication. Pipelined execution with push-only communication typically uses FIFO buffers to exchange results between operators (Harizopoulos et al., 2005). This allows to decouple query plans and have a distinct separation between queries, similar to a query-centric design. During SP, this forces the single thread of the pivot operator of the host packet to forward results to all satellite packets sequentially (see Figure 4.4a), which creates a serialization point.

This serialization point is reflected in the prediction model (Johnson et al., 2007), where the total work of the pivot operator includes a cost for forwarding results to all satellite packets. By using copying to forward results (Johnson et al., 2007), the serialization point becomes significant and delays consequent operators in the plans of the host and satellite packets. This creates a trade-off between sharing and parallelism, where in the latter case a query-centric model without sharing is used.

Sharing vs. Parallelism. We demonstrate this trade-off with the following experiment, similar to the basic experiment of (Johnson et al., 2007), which evaluates SP for the table scan stage with a memory-resident database. Though the trade-off applies for disk-resident databases and

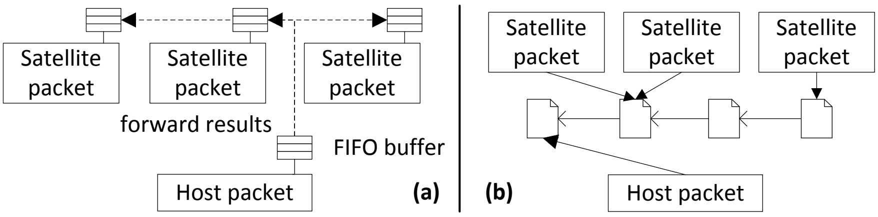


Figure 4.4: Sharing identical results during SP with: (a) push-only model and (b) a SPL.

other stages as well, it is more pronounced in this case. Our experimental configuration can be found in Section 4.5. We evaluate two variations of the QPipe execution engine: (a) No SP (FIFO), which evaluates query plans independently without any sharing, and (b) CS (FIFO), with SP enabled only for the table scan stage, thus supporting circular scans (CS). FIFO buffers are used for pipelined execution and copying is used to forward pages during SP, following the original push-only design (Harizopoulos et al., 2005; Johnson et al., 2007). We evaluate identical TPC-H (TPC, 2013) Q1 queries, submitted at the same time, with a database of scaling factor 1. In Figure 4.3a, we show the response times of the variations, while varying the number of concurrent queries.

For low concurrency, No SP (FIFO) efficiently uses available CPU resources. Starting with as few as 32 queries, however, there is contention for CPU resources and response times grow quickly, due to the fact that our server has 24 available cores and the query-centric model evaluates queries independently. For 64 queries it uses all cores at their maximum, resulting in excessive and unpredictable response times, with a standard deviation up to 30%.

CS (FIFO) suffers from low utilization of CPU resources, due to the aforementioned serialization point of SP. The critical path increases with the number of concurrent queries. For 64 queries, it uses an average of 3.1 of available cores. In this experiment, the proposed prediction model (Johnson et al., 2007) would not share in cases of low concurrency, essentially falling back to the line of No SP (FIFO), and would share in cases of high concurrency.

Nevertheless, the impact of the serialization point of SP can be minimized. Simply copying tuples in a multi-threaded way would not solve the problem, due to synchronization overhead and increased required CPU resources. A solution would be to forward tuples via pointers, a possibility not considered by the original system (Harizopoulos et al., 2005; Johnson et al., 2007). We can, however, avoid unnecessary pointer chasing; by employing pull-based communication, we can share the results and eliminate forwarding altogether. In essence, we transfer the responsibility of sharing the results from the producer to the consumers. Thus, the total work of the producer does not include any forwarding cost. Our pull-based communication model is adapted for SP for any stage with a step or linear WoP.

The serialization point of push-based SP was not a visible bottleneck in the original implementation, due to experiments being ran on a uni-processor (Harizopoulos et al., 2005). On multi-core machines its impact grows as the available parallelism increases, and a prediction model (Johnson et al., 2007) was proposed and used at run-time to dynamically decide whether (push-based) sharing should be employed. Our pull-based communication model for SP, however, addresses the serialization point by distributing the sharing cost, leading to better scaling on modern multi-core machines with virtually no sharing overhead.

To achieve this, we create an intermediate data structure, the ***Shared Pages Lists*** (SPL). SPL have the same usage as the FIFO buffers of the push-only model. A SPL, however, allows a single producer and multiple consumers. A SPL is a linked list of pages, depicted in Figure 4.4b. The producer adds pages at the head, and the consumers read the list from the tail up to the head independently.

In order to show the benefits of SPL, we run the experiment depicted in Figure 4.3a, by employing SPL instead of FIFO buffers. When SP does not take place, a SPL has the same role as a FIFO buffer, used by one producer and one consumer. Thus, the No SP (SPL) line has similar behavior with the No SP (FIFO) line. During SP, however, a single SPL is used to share the results of one producer with all consumers. In Figure 4.3b, we show the response times of the variations, while varying the number of concurrent queries.

With SPL, sharing has the same or better performance than not sharing, for all cases of concurrency. We avoid using a prediction model altogether, for deciding whether to share or not. Parallelism is achieved due to the minimization of the serialization point. For high concurrency, CS (SPL) uses more CPU resources and reduces response times by 82%-86% in comparison to CS (FIFO).

Additionally, in Figure 4.3c, we show the speedup of sharing over not sharing, for both models. We depict only values for low concurrency, as sharing is beneficial for both models in cases of high concurrency. We corroborate previous results on the negative impact of sharing with push-only communication (Johnson et al., 2007) for low concurrency, and show that pull-based sharing is always beneficial.

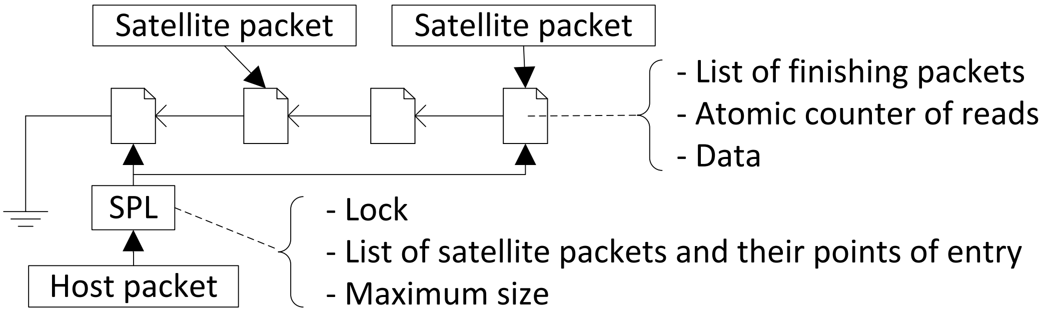

Satellite packet
Satellite packet
- List of finishing packets
- Atomic counter of reads
- Data
SPL
- Lock
- List of satellite packets and their points of entry
- Maximum size
Host packet

Figure 4.5: Design of a shared pages list.

4.3.1 Design of a SPL

In Figure 4.5, we depict a SPL. It points to the head and tail of the linked list. The host packet adds pages at the head. Satellite packets read pages from the SPL independently. Due to different concurrent actors accessing the SPL, we associate a lock with it. Contention for locking is minimal in all our experiments, mainly due to the granularity of pages we use (32KB). A lock-free linked list, however, can also be used to address any scalability problems.

Theoretically, if we allow the SPL to be unbounded, we can achieve the maximum parallelism possible, even if the producer and the consumers move at different speeds. There are practical reasons, however, why the SPL should not be unbounded, similar to the reasons why a FIFO buffer should not be unbounded, including: saving RAM, and regulating differently paced actors.

To investigate the effect of the maximum size of the SPL, we re-run the experiment of Figure 4.3b, for the case of 8 concurrent queries, varying the maximum size of SPL. In Figure 4.6, we show the response times. We observe that changing the maximum size of the SPL does not heavily affect performance. Hence, we chose a maximum size of 256KB for our experiments in Section 4.5 in order to minimize the memory footprint of SPL.

In order to decrease the size of the SPL, the last consumer is responsible for deleting the last page. Each page has an atomic counter with the number of consumers that will read this page. When a consumer finishes processing a page, he decrements its counter, deleting the page if he sees a zero counter. In order to know how many consumers will read a page, the SPL stores a list of active satellite packets. The producer assigns their number as the initial value of the atomic counter of each emitted page.

4.3.2 Linear Window of Opportunity

In order to handle a linear WoP, such as circular scans, the SPL stores the point of entry of every consumer. When the host packet finishes processing, the SPL is passed to the next host packet that handles the processing for re-producing missed results.

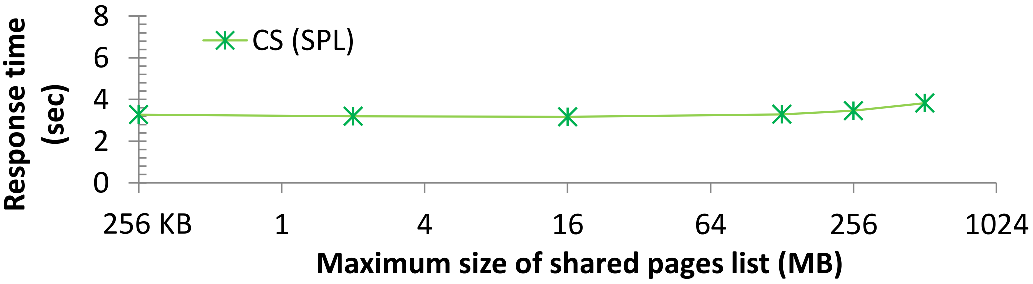


Figure 4.6: Same experiment as in Figure 4.3b, for the case of 8 concurrent queries, with a varying maximum size for SPL.

When the host packet emits a page, it checks for consumers whose point of entry is this page, and will need to finish when they reach it. The emitted page has attached to it a list of these finishing packets, which are removed from the active packets of the SPL (they do not participate in the atomic counter of subsequently emitted pages). When a consumer (packet) reads a page, it checks whether it is a finishing packet, in which case, it exits the SPL.

4.4 Experimental Methodology

We compare five variations of the QPipe execution engine:

- QPipe, without SP, which is similar to a typical query-centric model that evaluates query plans separately with pipelining, without any sharing. This serves as our baseline.
- QPipe-CS, supporting SP only for the table scan stage, i.e. circular scans (CS). It improves performance over QPipe by reducing contention for CPU resources, the buffer pool and the underlying storage device.
- QPipe-SP, supporting SP additionally for the join stage. It improves performance over QPipe-CS, in cases of high similarity, i.e. common sub-plans. In cases of low similarity, it behaves similar to QPipe-CS.
- CJOIN, without SP, which is the result of our integration of CJOIN into QPipe, hence the joins in star queries are evaluated with a GQP of shared hash-joins. We remind that CJOIN only supports shared hash-joins, thus subsequent operators are query-centric. Nevertheless, this configuration allows us to compare shared hash-joins with the query-centric ones used by the previous variations, giving us insight on the performance characteristics of general shared operators.
- CJOIN-SP, which additionally supports SP for the CJOIN stage (see Section 4.2.3). We use this configuration to evaluate the benefits of combining SP with a GQP. It behaves similar to CJOIN in cases of low similarity in the query mix.

In all our experiments, SP for the aggregation and sorting stages is off. This is done on purpose to isolate the benefits of SP for joins only, so as to better compare QPipe-SP and CJOIN-SP.

We use the Star Schema Benchmark (O'Neil et al., 2009) and Shore-MT (Johnson et al., 2009) as the storage manager. SSB is a simplified version of TPC-H (TPC, 2013) where the tables lineitem and order have been merged into lineorder and there are four dimension tables: date, supplier, customer and part. Shore-MT is an open-source multi-threaded storage manager developed to achieve scalability on multicore platforms.

Our server is a Sun Fire X4470 server with four hexa-core processors Intel Xeon E7530 at 1.86 GHz, with hyper-threading disabled and 64 GB of RAM. Each core has a 32KB L1 instructions cache, a 32KB L1 data cache, and a 256KB L2 cache. Each processor has a 12MB L3 cache, shared by all its cores. For storage, we use two 146 GB 10kRPM SAS 2.5” disks, configured as a RAID-0. The O/S is a 64-bit SMP Linux (Red Hat), with a 2.6.32 kernel.

We clear the file system caches before every measurement. All variations use a large buffer pool that fits SSB datasets of scaling factors up to 30 (corresponding to around 18GB of data produced by the SSB generator). SPL are used for exchanging results among packets. As we explain in Section 4.3, we use 32KB pages and a maximum size of 256KB for a SPL.

Unless stated otherwise, every data point is the average of multiple iterations with standard deviation less or equal to 5%. In some cases, contention for resources results in higher deviations. In certain experiments, we mention the average CPU usage and I/O throughput of representative iterations (averaged only over their activity period), to gain insight on the performance of the variations.

We vary (a) the number of concurrent queries, (b) whether the database is memory-resident or disk-resident, (c) the selectivity of fact tuples, (e) the scaling factor, and (d) the query similarity which is modeled in our experiments by the number of possible different submitted query plans. Queries are submitted at the same time, and are all evaluated concurrently. This single batch for all queries allows us to minimize query admission overheads for CJOIN, and additionally allows us to show the effects of SP, as all queries with common sub-plans arrive surely inside the WoP of their pivot operators. We note that variable inter-arrival delays can decrease sharing opportunities for SP, and refer the interested reader to the original QPipe paper (Harizopoulos et al., 2005) to review the effects of inter-arrival delays for different cases of pivot operators and WoP.

Finally, in Section 4.5.4, we evaluate QPipe-SP, CJOIN-SP, and Postgres with a mix of SSB queries and a scaling factor 30. We use PostgreSQL 9.1.4 as another example of a query-centric execution engine that does not share among concurrent queries. We configure PostgreSQL to make the comparison with QPipe as fair as possible. We use 32KB pages, large shared buffers that fit the database, ensure that it never spills to the disk and that the query execution plans are the same. We disable query result caching, which does not execute a previously seen query at all. We do not want to compare caching of previously executed queries, but the efficiency of sharing among in-progress queries.

4.5 Experimental Analysis

In this section, we measure performance by evaluating multiple concurrent instances of SSB Q3.2. It is a typical star query that joins three of the four dimension tables with the fact table. The SQL template and the execution plan are shown in Figure 4.7. We selected a single query template for our sensitivity analysis because we can adjust the similarity of the query mix to gain insight on the benefits of SP, and also, the GQP of CJOIN is the same for all experiments, with 3 shared hash-joins for all star queries.

```sql
SELECT    c_city, s_city, d_year,
          SUM(lo_revenue) as revenue
FROM      customer, lineorder, supplier, date
WHERE     lo_custkey = c_custkey
          AND lo_suppkey = s_suppkey
          AND lo_orderdate = d_datekey
          AND c_nation = [NationCustomer]
          AND s_nation = [NationSupplier]
          AND d_year >= [YearLow]
          AND d_year <= [YearHigh]
GROUP BY c_city, s_city, d_year
ORDER BY d_year ASC, revenue DESC
```

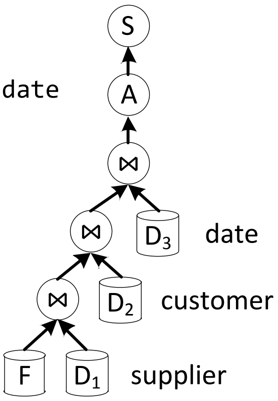
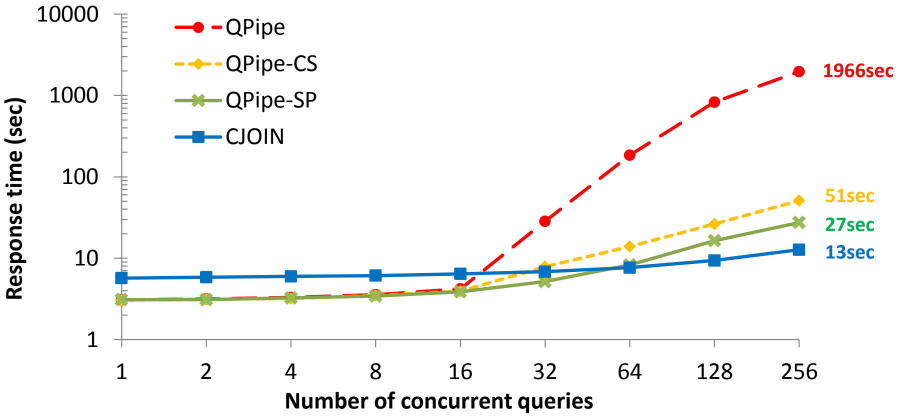


Figure 4.7: The SSB Q3.2 SQL template and the query plan.



Measurement \Variation	QPipe	QPipe-CS	QPipe-SP	CJOIN
Avg. # Cores Used	23.91	19.72	18.75	3.47

Figure 4.8: Memory-resident database of SF=1. The table includes measurements for the case of 256 queries.

4.5.1 Impact of concurrency

We start with an experiment that does not involve I/O accesses in order to study the computational behavior of the variations. We store our database in a RAM drive. We evaluate multiple concurrent SSB Q3.2 instances for a scaling factor 1. The predicates of the queries are chosen randomly, keeping a low similarity factor among queries and the selectivity of fact tuples varies from 0.02% to 0.16% per query. In Figure 4.8, we show the response times of the variations, while varying the number of concurrent queries.

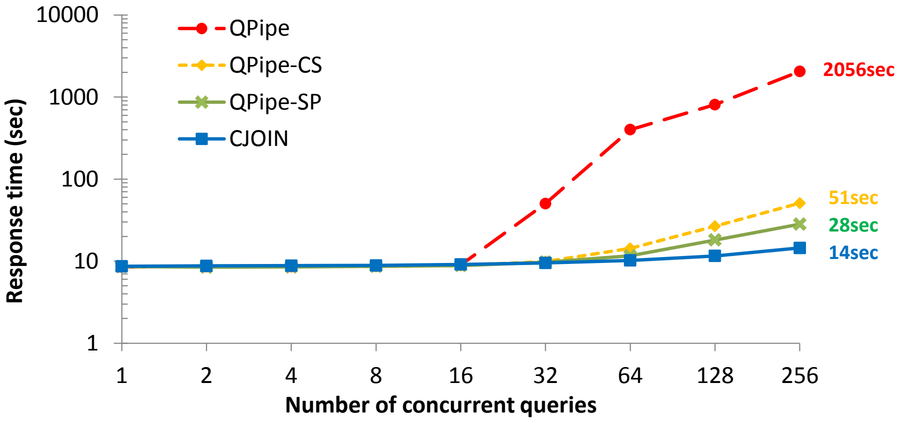
For low concurrency, QPipe successfully uses available CPU resources. Starting with as few as 32 concurrent queries, there is contention for CPU resources, due to the fact that our server has 24 cores and QPipe evaluates queries separately. Response times grow quickly and unpredictability results in standard deviations up to 50%. For 256 queries it uses all cores at their maximum.

The circular scans of QPipe-CS reduce contention for CPU resources and the buffer pool, improving performance. For high concurrency, however, there are more threads than available hardware contexts, thus increasing response times.

QPipe-CS misses several sharing opportunities at higher operators in the query plans. QPipe-SP can exploit them. Even though we use random predicates, the ranges of variables of the SSB Q3.2 template allows QPipe-SP to share the first hash-join 126 times, the second hash-join 17 times, and the third hash-join 1 time, on average for 256 queries. Thus, it saves more CPU resources, and results in lower response time than the circular scans alone.

The shared operators of CJOIN offer the best performance, as they are the most efficient in saving resources. CJOIN has an initialization overhead in comparison to the other variations, attributed to its admission phase, which takes place before the evaluation (see Section 4.2.1). These costs are low and the shared hash-joins in the GQP can effortlessly evaluate many instances of SSB Q3.2. Nevertheless, admission and evaluation costs accumulate for an increasing number of queries, thus the CJOIN line also starts to degrade.

We do not depict CJOIN-SP, as it has the same behavior as CJOIN. As we noted in Section 4.2.3, our implementation of CJOIN-SP supports sharing CJOIN packets with all join predicates identical. This is rare due to this experiment's random predicates.



Measurement \Variation	QPipe	QPipe-CS	QPipe-SP	CJOIN
Avg. # Cores Used	23.86	19.84	17.06	3.49
Avg. Read Rate (MB/s)	1.88	74.47	97.67	156.11

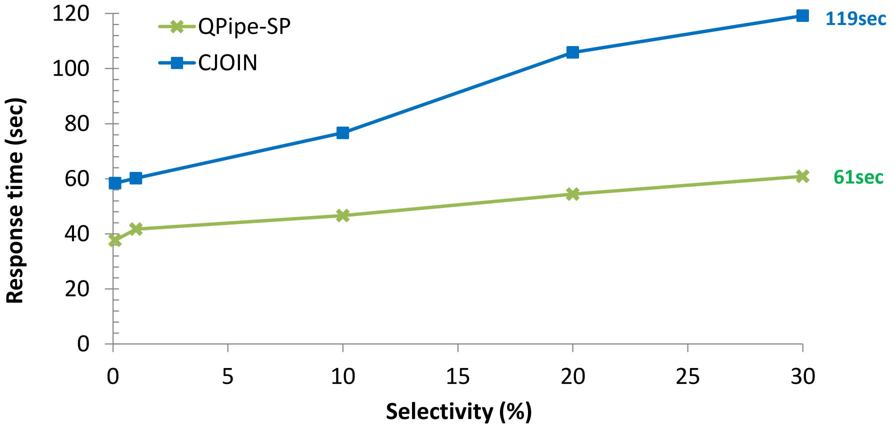
Figure 4.9: Same as Figure 4.8, with a disk-resident database.

Our observations apply also for disk-resident databases. In Figure 4.9, we show the results of the same experiment with the database on disk. QPipe suffers from CPU contention, which de-schedules scanner threads regularly resulting in low I/O throughput. Additionally, scanner threads compete for bringing pages into the buffer pool. Response times for low concurrency have increased, but not significantly for high concurrency, due to the fact that the workload is CPU-bound for high concurrency. QPipe-CS improves performance (by 80%-97% for high concurrency) by reducing contention for resources and the buffer pool. QPipe-SP further improves performance by eliminating common sub-plans. The shared operators of CJOIN still prevail for high concurrency. Furthermore, we note that the overhead of the admission phase, that we observed for a memory-resident database, is masked by file system caches for disk-resident databases. We explore this effect in a next experiment, where we vary the scaling factor.

Implications. Shared scans improve performance by reducing contention for resources, the buffer pool and the underlying storage devices. SP is able to eliminate common sub-plans. Shared operators in a GQP are more efficient in evaluating a high number of queries, in comparison to standard query-centric operators.

4.5.2 Impact of data size

In this section, we study the behavior of the variations by varying the amount of data they need to handle. We perform two experiments: In the first, we vary the selectivity of fact tuples of queries, and in the second, the scaling factor.

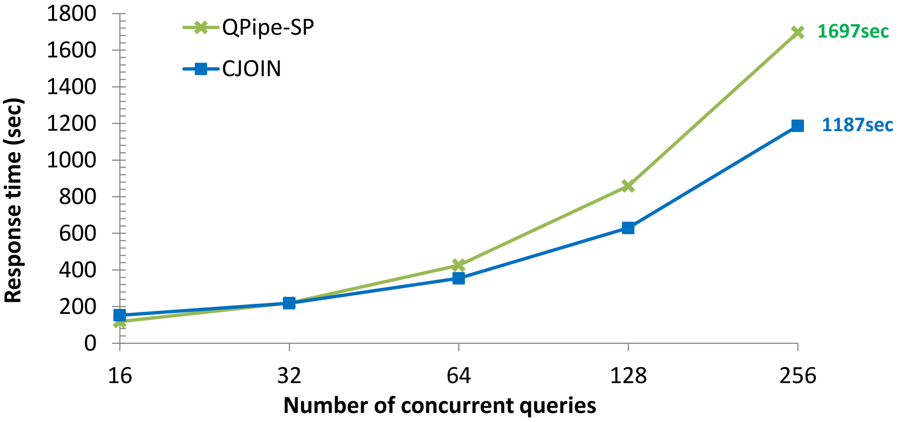


Measurement \Variation	QPipe-SP	CJOIN
Avg. # Cores Used	17.79	18.86

Figure 4.10: 8 queries with a memory-resident database of SF=10. The table includes measurements for 30% selectivity.

Impact of selectivity. We use a memory-resident database with scaling factor 10. The query mix consists of 8 concurrent queries which are different instances of a modified SSB Q3.2 template. The template is based on Q3.2, but for the year range we select the maximum possible range. Moreover, we extend the WHERE clause of the query template by adding more options for both customer and supplier nation attributes. For example, if we use a disjunction of 2 nations for customers and 3 nations for suppliers, we achieve a selectivity of $\frac{2}{25}\frac{3}{25}\frac{7}{7} \approx 1\%$ of fact tuples. Nations are selected randomly over all 25 possible values and are unique in every disjunction, keeping a minimal similarity factor. The results are shown in Figure 4.10. In this experiment, there is no contention for resources and no common sub-plans. Thus, we do not depict QPipe and QPipe-CS, as they have the same behavior as QPipe-SP, and we do not depict CJOIN-SP, as it has the same behavior as CJOIN.

Both QPipe-SP and CJOIN show a degradation in performance as selectivity increases, due to the increasing amount of data they need to handle. Their trends, however, are different. The response time of CJOIN increases more quickly than the response time of QPipe-SP. This is due to two facts. Firstly, the admission phase of CJOIN is extended, as it evaluates more complex selection predicates for all queries, whereas in circular scans, queries evaluate their predicates independently. Secondly, and more importantly, the shared operators of a GQP inherently entail a book-keeping overhead, in comparison to standard query-centric operators. In our case, the additional cost of shared hash-joins includes (a) the maintenance of larger hash tables for the union of the selected dimension tuples of all concurrent queries, and (b) bit-wise AND operations between the bitmaps of tuples. Query-centric operators do not entail these costs, and maintain a hash table for one query.



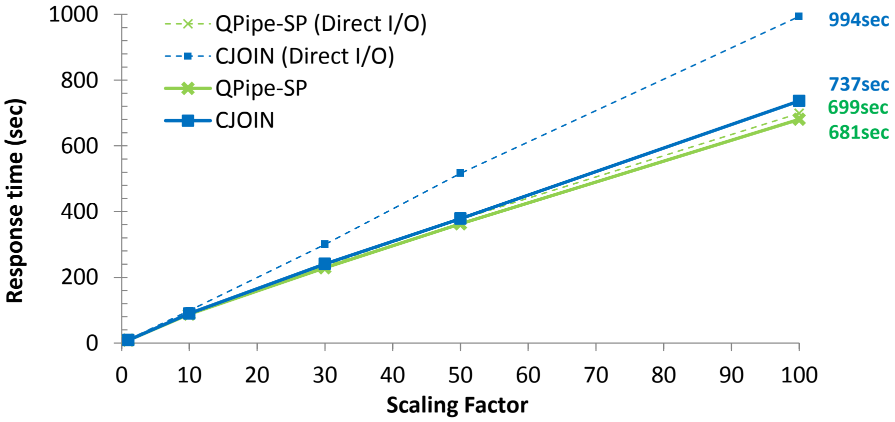
Measurement \Variation	QPipe-SP	CJOIN
Avg. # Cores Used	22.86	17.73

Figure 4.11: Memory-resident database of SF=10 and 30% selectivity. The table includes measurements for 256 queries.

The book-keeping overhead can be decreased significantly with careful implementation choices. DataPath (Arumugam et al., 2010) uses a single large hash table for all shared hash-joins, decreasing the maintenance and access costs for hash tables. Nevertheless, shared hash-joins still need to process the union of selected tuples and perform bit-wise operations. The same reasoning can be applied to other shared operators as well, such as a shared aggregations. A single shared aggregation maintains a running sum for all queries. For low concurrency, it reduces parallelization in comparison to standard query-centric aggregations that calculate a running sum for each query.

We used 8 queries to avoid CPU contention. For higher concurrency, shared operators still prevail, due to their efficiency in saving resources. We depict in Figure 4.11 the response times for the case of 30% selectivity. For high concurrency, the query-centric operators of QPipe-SP contend for CPU resources. CJOIN is able to save more resources and outperform the query-centric operators.

Impact of Scaling Factor. We show the same trade-off between shared operators and query-centric operators by varying the scaling factor. We use disk-resident databases and 8 concurrent queries with random predicates and selectivity between 0.02% and 0.16%. The results are shown in Figure 4.12. The response times of both QPipe-SP and CJOIN increase linearly. Their slopes, however, are different. The reasons are the same: The admission phase is extended, and shared operators entail a book-keeping overhead.



Measur.\Variations	QPipe-SP (Direct I/O)	CJOIN (Direct I/O)	QPipe-SP	CJOIN
# Cores Used	5.96	1.68	5.38	2.47
Read Rate (MB/s)	97.16	70.01	215.58	204.71

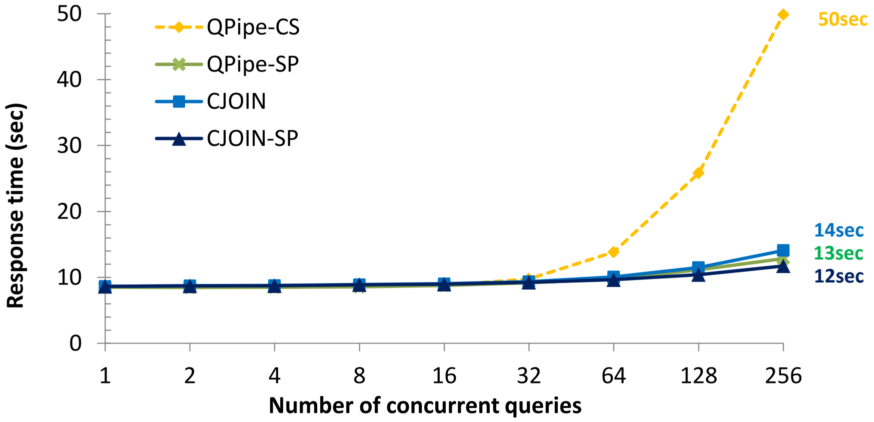
Figure 4.12: 8 concurrent queries with disk-resident databases. The table includes measurements for the case of SF=100.

We also show the response time of the two variations by using direct I/O for accessing the database on disk, to bypass file system caches. This allows us to isolate the overhead of CJOIN's preprocessor. As we have mentioned before, the preprocessor is in charge of the circular scan of the fact table, the admission phase of new queries, and finalizing queries when they wrap around to their point of entry. These responsibilities slow the circular scan significantly. Without direct I/O, file system caches coalesce contiguous I/O accesses and read-ahead, achieving high I/O read throughput in sequential scans, masking the preprocessor's overhead.

Implications. For low concurrency, a GQP with shared operators entails a book-keeping overhead in comparison to query-centric operators. For high concurrency, however, shared operators prevail due to their efficiency in saving resources.

4.5.3 Impact of Similarity

In this experiment we use a disk-resident database of scaling factor 1. We limit the randomness of the predicates of queries to a small set of values. Specifically, there are 16 possible query plans for instances of Q3.2. The selectivity of fact tuples ranges from 0.02% to 0.05%. In Figure 4.13, we show the response times of the variations, varying the number of concurrent queries. We do not depict QPipe, as it does not exploit any sharing and results in increased contention and high response times for high concurrency. We remind that we do not enable SP for aggregations and sorting, in order to focus on the effect of SP for joins.



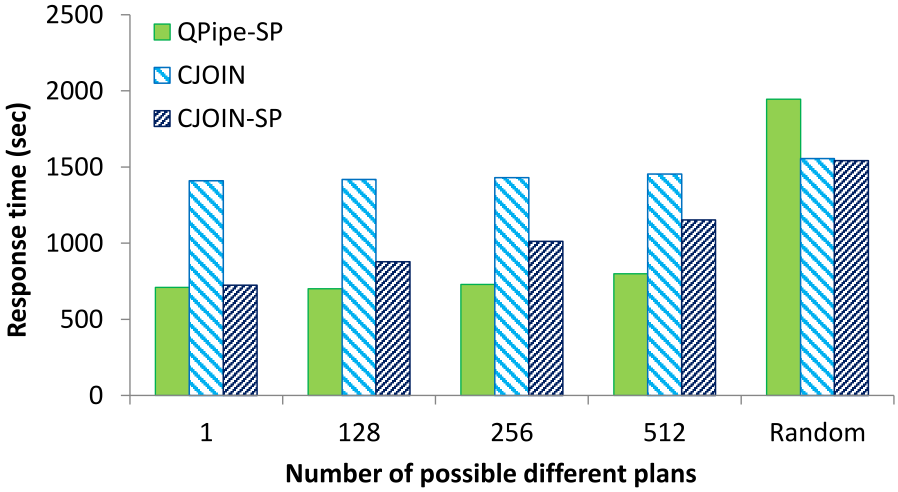
Measur. \Variation	QPipe-CS	QPipe-SP	CJOIN	CJOIN-SP
Avg. # Cores Used	20.32	9.60	2.50	2.34
Avg. Read Rate (MB/s)	74.37	120.20	130.18	130.15

Figure 4.13: Disk-resident database of SF=1 and 16 possible plans. The table includes measurements for 256 queries.

QPipe-SP evaluates a maximum of 16 different plans and re-uses results for the rest of similar queries. It shares the second hash-join 1 time, and the third hash-join 238 times, on average, for 256 queries. This leads to high sharing and minimal contention for computations. On the other hand, QPipe-CS does not share operators other than the table scan, resulting in high contention.

Similarly, CJOIN misses exploiting these sharing opportunities and evaluates identical queries redundantly. In fact, QPipe-SP outperforms CJOIN. CJOIN-SP, however, is able to exploit them. For a group of identical star queries, only one is evaluated by the GQP. CJOIN-SP shares CJOIN packets 239 times on average for 256 queries. Thus, CJOIN-SP outperforms all variations.

To further magnify the impact of SP, we perform another experiment for 512 concurrent queries, a scaling factor of 100 (with a buffer pool fitting 10% of the database), and varying the number of possible different query plans. The results are shown in Figure 4.14. CJOIN is not heavily affected by the number of different plans. For the extreme cases of high similarity, QPipe-SP prevails over CJOIN. For lower similarity, the number of different plans it needs to evaluate is



Differ. plans	1	128	256	512	Random
QPipe-SP	1/0/510	18/94/381	49/156/287	106/196/188	362/82/5
CJOIN-SP	510	384	287	190	12

Figure 4.14: Evaluating 512 concurrent queries with a varying similarity factor, for a SF=100. The table includes the SP sharing opportunities (average of all iterations), in the format 1st/2nd/3rd hash-join for QPipe-SP.

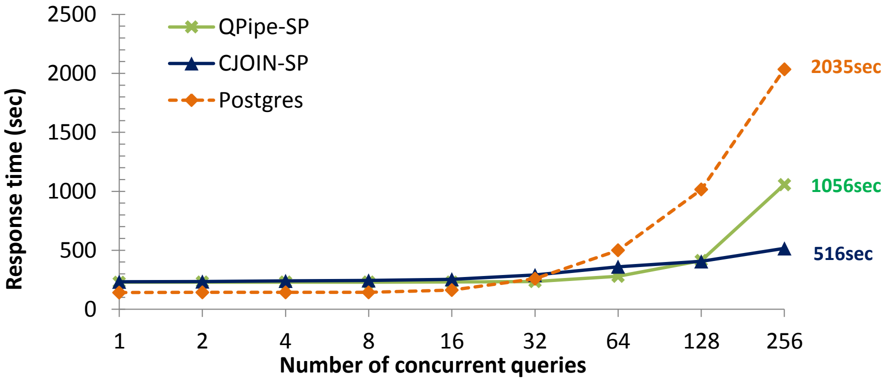
larger and performance is deteriorated due to contention for resources. CJOIN-SP is able to exploit identical CJOIN packets and improve performance of CJOIN by 20%-48% for cases with common sub-plans in the query mix.

Implications. We can combine SP with a GQP to eliminate redundant computations and improve performance of shared operators for a query mix that exposes common sub-plans.

4.5.4 SSB query mix evaluation

In this section we evaluate QPipe-SP, CJOIN-SP, and Postgres using a mix of three SSB queries (namely Q1.1, Q2.1 and Q3.2), with a disk-resident database and a scaling factor of 30. The predicates for the queries are selected randomly and the selectivity of fact tuples is less than 1%. Each query is instantiated from the three query templates in a round-robin fashion, so all variations contain the same number of instances for each query type.

In Figure 4.15, we depict the response times of the variations, while varying the number of concurrent queries. As Postgres is a more mature system than the two research prototypes, it attains a better performance for low concurrency. Our aim, however, is not to compare the per-query performance of the variations, but their efficiency in sharing among a high number of concurrent queries. Postgres follows a traditional query-centric model of execution, and does not share among in-progress queries. For this reason, it results in contention for resources.



Measurement \Variation	Postgres	QPipe-SP	CJOIN-SP
Avg. # Cores Used	18.56	19.07	19.11
Avg. Read Rate (MB/s)	15.93	84.98	110.03

Figure 4.15: Disk-resident database of SF=30. The table includes measurements for the case of 256 queries.

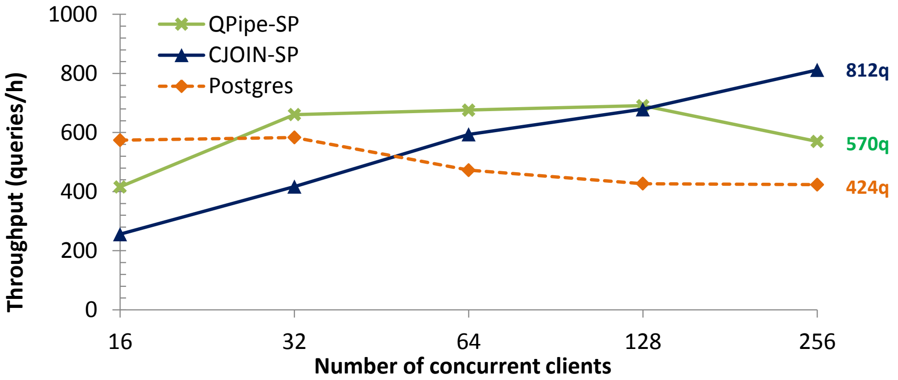
QPipe-SP results in a better performance due to circular scans and the elimination of common sub-plans. CJOIN-SP attains the best performance, as shared operators are the most efficient in sharing among concurrent queries.

In Figure 4.16, we show the throughput of the three variations, by varying the number of concurrent clients. Each client initially submits a query, and when it finishes, the next one is submitted. The shared operators of a GQP are able to handle new queries with minimal additional resources. Thus, the throughput of CJOIN-SP increases almost linearly. The throughput of the query-centric operators of Postgres and QPipe-SP, however, ultimately decreases with an increasing number of clients, due to resources contention.

4.6 Discussion

Shared scans and SPL. Pull-based models, similar to SPL, have been proposed for shared scans that are specialized for efficient buffer pool management and are based on the fact that all data are available for accessing (see Section 2.3). SPL differ because they are generic and can be used during SP at any operator which may be producing results at run-time. It is possible, however, to use shared scans for table scans, and use SPL during SP for other operators.

Prediction model for sharing with a GQP. Shared operators of a GQP are not beneficial for low concurrency, in comparison to the query-centric model, because they entail an increased book-keeping overhead (see Section 4.5.2). The turning point, however, when shared operators become beneficial needs to be pinpointed. A simple heuristic is the point when resources become saturated (see Table 4.1). An exact solution would be a prediction model, similar to (Johnson



Measurement \Variation	Postgres	QPipe-SP	CJOIN-SP
Avg. # Cores Used	18.29	19.59	13.70
Avg. Read Rate (MB/s)	15.94	67.42	79.98

Figure 4.16: Throughput while evaluating concurrent queries with a disk-resident database of SF=30. The table includes measurements for the case of 256 clients.

et al., 2007). This model, however, targets only sharing identical results during SP (see Section 4.3). Shared operators in a GQP do not share identical results, but part of their evaluation among possibly different queries. A potential prediction model for a GQP needs to consider the book-keeping overhead, and the cost of optimizing the GQP, for the current query mix and resources.

Distributed environments. Our work focuses on scaling up and not out. Following prior work (Arumugam et al., 2010; Candea et al., 2009; Giannikis et al., 2012; Harizopoulos et al., 2005), we consider scaling up as a base case, because it is a standard means of increasing throughput in DBMS. We intend to follow up with further research in parallel DBMS (Mehta et al., 1993) and other distributed data systems (Dean and Ghemawat, 2004). For example, we can improve global scheduling in parallel DBMS by taking sharing into account: each replica node can employ a separate GQP, and a new query should be dispatched to the node which will incur the minimum estimated marginal cost for evaluation.

4.7 Conclusions

In this chapter we perform an experimental study to answer *when* and *how* an execution engine should share data and work across concurrent analytical queries. We review work sharing methodologies and we study Simultaneous Pipelining (SP) and Global Query Plans with shared operators (GQP) as two state-of-the-art sharing techniques. We perform an extensive evaluation of SP and GQP, based on their original research prototype systems.

Work sharing is typically beneficial for high concurrency because the opportunities for common work increase, and it reduces contention for resources. For low concurrency, however, there is a trade-off between sharing and parallelism, particularly when the sharing overhead is significant. We show that GQP are not beneficial for low concurrency because shared operators inherently involve an additional book-keeping overhead, compared to query-centric operators. For SP, however, we show that it can be beneficial for low concurrency as well, if the appropriate communication model is employed: we introduce SPL, a pull-based approach for SP that scales better on modern multi-core machines than push-based SP. SPL is a data structure that naturally promotes parallelism by shifting the responsibility of sharing common results from the producer to the consumers.

Furthermore, we show that SP and GQP are two orthogonal sharing techniques and their integration allows to share operators and handle a high number of concurrent queries, while also sharing any common sub-plans presented in the query mix. In conclusion, analytical query engines should dynamically choose between query-centric operators with SP for low concurrency and GQP with shared operators enhanced by SP for high concurrency.

5 BF-Tree: Approximate Tree Indexing[1]

5.1 Introduction

Database Management Systems (DBMS) have been traditionally designed with the assumption that the underlying storage is comprised of hard disks (HDD). This assumption impacts most of the design choices of DBMS and in particular the ones of the storage and the indexing subsystems. Data stored on the secondary storage of a DBMS can be accessed either by a sequential scan or by using an index for randomly located searches. The most common types of indexes in modern database systems are B^+-Trees and hash indexes (Ramakrishnan and Gehrke, 2002). Other types of indexing include bitmap indexes (O'Neil, 1987).

Tree structures like B^+-Trees are widely used because they are optimized for the common storage technology - HDD - and they offer efficient indexing and accessing for both sorted and unsorted data (e.g. in heap files). Tree structures offer logarithmic, to the size of the data, number of random accesses (and, as a result, lookup time) and support ordered range scans. Hash tables are very efficient for point queries, i.e. for a single value probe, because, once hashing is completed the search cost is constant. Indexing in today's systems is particularly important because it needs only a few random accesses to locate any value, which is sublinear to the size of the data (e.g., logarithmic for trees, constant for hash indexes).

5.1.1 Implicit Clustering

Big data analysis - often performed in a real-time manner - is becoming increasingly more popular and crucial to business operation. New datasets including data from scientists (e.g., simulations, large-scale experiments and measurements, sensor data), social data (e.g., social status updates, tweets) and, monitoring, archival and historical data (managed by data warehousing systems), all have a time dimension, leading to implicit, time-based clusters of data, often resulting in

---

[1] The material of this chapter has been the basis for a paper submitted for publication entitled "BF-Tree: Approximate Tree Indexing" (Athanassoulis and Ailamaki, 2013).

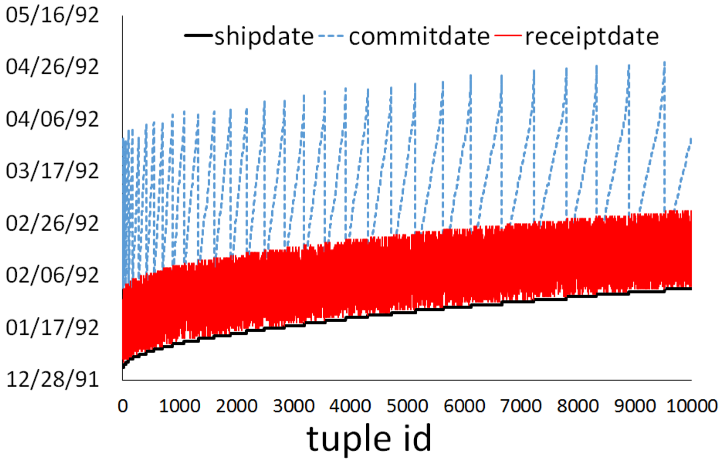


Figure 5.1: Clustering of shipdate, commitdate, and receiptdate.

storing data based on the creation timestamp. The property of *implicit clustering* (Moerkotte, 1998) characterizes data warehousing datasets, which are often naturally partitioned for the attributes that are correlated with time. For example, in a typical data warehousing benchmark (TPCH (TPC, 2013)) for every purchase we have three dates (ship date, commit date and receipt date). While these three dates do not have the same order for different products, the variations are small and the three values are typically close. Figure 5.1 shows the dates of the first 10000 tuples of the lineitem table of the TPCH benchmark when data are ordered using the creation time.

Today, real-time applications like energy smart meters (IBM, 2012) and Facebook (Borthakur, 2013) have a constant ingest of timestamped data. In addition to the real-time nature of such applications, immutable files with historical time-generated data are stored and the goal is to offer cheap yet efficient storage and searching. For example, Facebook has announced projects to offer cold storage (Taylor, 2013) using flash or shingled disks (Hughes, 2013). When cold data are stored on low-end flash chips as immutable files, they can be ordered or partitioned anticipating future access patterns, offering explicit clustering.

Datasets with either implicit or explicit clustering are *ordered* or *partitioned*, typically, on a time dimension. In this chapter, we design an index that is able to exploit such data organization to offer competitive search performance with smaller index size.

5.1.2 The capacity-performance trade-off

Solid-state disks (SSD) use technologies like flash and Phase Change Memory (PCM) (Doller, 2009) that do not suffer from mechanical limitations like *rotational delay* and *seek time*. They have virtually the same random and sequential read throughput and several orders of magnitudes smaller read latency when compared to hard disks (Bouganim et al., 2009; Stoica et al., 2009). The capacity of SSD, however, is a scarce resource compared with the capacity of HDD. Typically, SSD capacity is one order of magnitude more expensive than HDD capacity. The contrast between capacity and performance creates a *storage trade-off*. Figure 5.2 shows how several SSD and

HDD devices (as of end 2013) are characterized in the trade-off according to their capacity (GB per $) on the x-axis and the advertised random read performance (IOPS) on the y-axis. We show two enterprise-level and two consumer-level HDD (E- and C-HDD respectively) and, four enterprise-level and two consumer-level SSD (E- and C-SSD respectively). The two technologies create two distinct clusters. HDD are on the lower right part of the figure offering cheap capacity (and in all cases cheaper than SSD) and inferior performance - in terms of random read I/O per second (IOPS) - varying from one to four orders of magnitude. Hence, instead of designing indexes for storage with cheap capacity and expensive random accesses (for HDD), today we need to design indexes for storage with fast random accesses with expensive capacity (for SSD).

5.1.3 Indexing for modern storage

Systems and applications requirements are heavily impacted by the emergence of SSD and there has been a plethora of research aiming at integrating and exploiting such devices with existing DBMS. These efforts include flash-only DBMS (Stoica et al., 2009; Tsirogiannis et al., 2009), flash-HDD hybrid DBMS (Koltsidas and Viglas, 2008), using flash in a specialized way (Athanassoulis et al., 2011; Chen, 2009) and optimizing internal structures of the DBMS for flash (e.g., flash-aware B-Trees and write-ahead-logging (Agrawal et al., 2009; Chen, 2009; Gao et al., 2011; Li et al., 2010; Roh et al., 2011)). Additionally, the trends of increasing capacity and performance of SSD lead to higher adoption of hybrid or flash-only storage subsystems (Borthakur, 2013). Thus, more and more data reside on SSD and we need to access them in an efficient way.

SSD-aware indexes are not enough. Prior approaches for flash-aware B^+-Tree, however, focus on addressing the slow writes on flash and the limited device lifetime (see Section 2.5). These techniques do not address the aforementioned *storage trade-off* - between storage capacity and performance - since they follow the same principles that B^+-Trees are built with: minimize the number of slow random accesses by having a wide (and potentially large) tree structure. Instead of decreasing the index size at the expense of more random reads, traditional tree indexing uses larger size in order to reduce random reads.

5.1.4 Approximate Tree Indexing

We propose a novel form of sparse indexing, using an approximate indexing technique which leverages efficient random reads to offer performance competitive with traditional tree indexes, reducing drastically the index size. The smaller size enables fast rebuilds if needed. We achieve this by indexing in a lossy manner and exploiting natural partitioning of the data in a Bloom filter tree (BF-Tree). In the context of BF-Trees, Bloom filters are used to store the information whether a key exists in a specific range of pages. BF-Trees can be used to index attributes that are ordered or naturally partitioned within the data file. Similarly to a B^+-Tree, a BF-Tree has two types of nodes: internal and leaf nodes. The internal nodes resemble the ones of a B^+-Tree

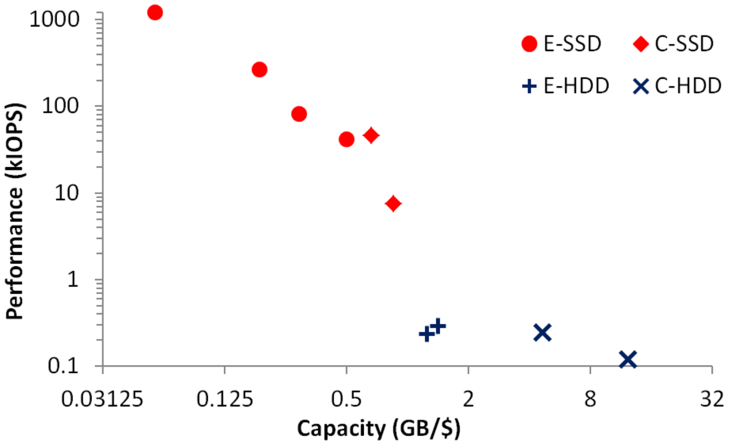


Figure 5.2: The capacity/performance storage trade-off.

but the leaf nodes are radically different. A leaf node of a BF-tree (BF-leaf) consists of one or more Bloom filters which store the information whether a key for the indexed attribute exists in a particular range of data pages. The choice of Bloom filters as the building block of a BF-Tree allows accuracy parametrization based on (i) the indexing granularity (data pages per Bloom filter) and (ii) the indexing accuracy (false positive probability of the Bloom filters). The former is useful when the data are not strictly ordered and the latter can be used to vary the overall size of the tree.

Contributions. This chapter makes the following contributions:

- We identify the *capacity-performance storage trade-off*.
- We introduce approximate tree indexing using probabilistic data structures, which can be parametrized to favor either accuracy or capacity. We present such an index, the BF-Tree, which is designed for workloads with implicit clustering tailored for emerging storage technologies.
- We model the behavior of BF-Trees and present a study that analytically predicts their performance and compares with B$^+$-Trees.
- We show in our experimental analysis that BF-Trees offer competitive performance with 2.22x to 48x smaller index size when compared with B$^+$-Trees and hash indexes.

Outline. In Section 5.2 we present a key insight which makes BF-Trees viable, in Sections 5.3 and 5.4 we present the internals of a BF-Tree. Section 5.5 presents an analytical model predicting BF-Trees behavior and Section 5.6 presents the evaluation of BF-Trees. In Section 5.7 we further discuss BF-trees as a general index and in Section 5.8 we discuss additional optimizations that BF-Tree supports. Finally, Section 5.9 concludes this chapter.

5.2 Making approximate indexing competitive

Bloom proposed (Bloom, 1970) BF as a space-efficient probabilistic data structure which supports membership tests, with a false positive probability. On the other hand, the small size of a BF allows for its re-computation when needed. Thus, in BF-Trees we do not employ a BF for the entire relation. We use BF to perform a membership test for a specific range, keeping the range of values for a single BF small in order to be feasible to recompute it.

Here, we extend the background discussion for BF of Section 2.4 by focusing on a key property used to optimize indexing partitioned or ordered data. A BF is comprised of m bits, and it stores membership information for n elements with false positive probability p. An empty BF is an array of m bits all set to 0. We need, as well, k different hash functions to be used to map an element to k different bits during the process of adding an element or checking for membership. When an element is added, the k hash functions are used in order to compute which k out of m bits have to be set to 1. If a bit is already 1 it maintains this value. To test an element for membership the same k bits are read and, if any of the k bits is equal to 0, then the element is not in the set. If all k bits are equal to 1 then the element belongs to the set with probability $1-p$. Assuming optimal number of k hash functions the connection between the BF parameters is approximated by the formula[2] (Tarkoma et al., 2012):

$$n = -\frac{m \cdot ln^2(2)}{ln(p)} \tag{5.1}$$

From this formula we can derive some useful properties:

1. If a BF with size M bits can store the membership information of N elements with false positive p, then S BFs with size $\frac{M}{S}$ bits each can store the membership information of $\frac{N}{S}$ elements each with the same p.
2. Decreasing the probability of false positives has a logarithmic effect on the number of elements we can index using a given space budget (i.e., number of bits).

Property (1) allows to divide the index into smaller BFs that incorporate location information. This process is done hierarchically and is presented in Section 5.3 and Section 5.4. We present a tree structure with a large BF per leaf, which contains membership information, and internal nodes which help navigating to the desired range before we do the membership test. Each leaf node corresponds to a number of data pages and upon positive membership test we have to search these data pages for the desired values. In Section 5.4 we describe how we can store location information more efficiently by storing one – small – BF per data page in the leaf nodes. The leaf node now consists of several BFs which can be checked in parallel and produce the list of data pages to be read more efficiently.

---

[2] When modelling the behavior of a BF we will use the given formula unless otherwise stated.

5.3 Bloom Filter Tree (BF-Tree)

In this section we describe in detail the structure of a BF-tree, highlighting its differences from a typical B^+-Tree.

5.3.1 BF-Tree architecture

A BF-tree consists of nodes of two different types. The root and the internal nodes have the same morphology as a typical B^+-Tree node: they contain a list of keys with pointers to other nodes between each pair of keys (see Figure 5.3). If the referenced node is internal then it has exactly the same structure. The leaf nodes (BF-leaves), however, are different. Each leaf node contains a *BF* which can be used to test any key in the corresponding *range* of the node, for key membership. Finally, each leaf node contains a *list of pairs <key, page id>* which is used to enhance the search, called *run indexes*. The run indexes serve as barriers dividing a range of pages in partitions based on the key value, hence, both *keys* and *page ids* are ordered. Note that the data does not need to be ordered for a partition scheme to hold, since the run indexes serve as fences. The assumption is that, for each BF-leaf, we know either implicitly or explicitly a sub-range of pages that we need to search in if the key in question is found to exist according to the BF. This data organization exists when the data are ordered on the indexing key or partitioned using the indexing key. In cases of composite indexing keys, or for attributes that have values correlated with the order of the data, such an assumption can hold for more than one index.

BF-leaf. For simplicity and compatibility with the existing framework, the root, the internal nodes and the leaf nodes have the same size (typically either 4KB or 8KB, we will assume 4KB in the following discussion). The BF of a BF-leaf contains the membership information of the values of a specific key in a range of pages. Assuming that the run indexes of a given BF-leaf are $< k_1, p_1 >, \ldots, < k_n, p_n >$ the membership test will answer whether a key which has value in the interval $[k_1, k_n]$ exists within the range of pages $p_1, \ldots, p_n$. The run indexes can further assist the search once the membership test is positive to eliminate searching in pages which contain keys out of the desired range. In every BF-leaf (Figure 5.4(a)) we store the following information:

- The smallest key of the node's BF: *min_key*.
- The largest key of the node's BF: *max_key*.
- The number of currently indexed elements: *#keys*.
- A pointer to the next BF[3]: *next_BFleaf*.
- The *actual BF* to perform the membership test.
- A list of keys and page ids which partition the corresponding contiguous pages into smaller sets of pages that contain sub-ranges of the leaf's range: *run indexes*.

---

[3] Useful for bulk loading, the pointers between BF-Tree leaf nodes are similar to pointers between B^+-Trees leaf nodes.

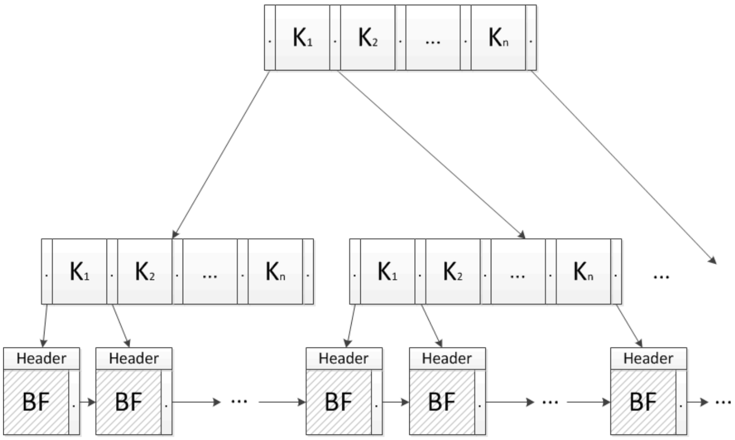


Figure 5.3: BF-tree

The search time of a BF-tree depends on three parameters:

- The height of the tree.
- The range of pages corresponding to each BF-leaf, which depends on the *false positive probability* (fpp).
- The number of run indexes per BF-leaf.

Searching for a tuple. As shown in Algorithm 1, once we have retrieved the desired BF-leaf, we perform a membership test in the BF. This test decides whether the key we search for exists in our dataset with probability for false positive answer fpp. The average search cost includes the overhead of false positives, which is negligible when fpp is low as we show in Sections 5.5 and 5.6. Once we have a positive answer we have to perform a binary search on the run indexes and start from the page closer to the desired key. Following the assumption that the data are either ordered or partitioned by the index key, we have finally to search in the entire range of pages of the BF-leaf before we can be sure whether the tuple exists or it was a false positive.

5.3.2 Creating and Updating a BF-Tree

For creating and updating a BF-Tree, the high-level B^+-Tree algorithms are still relevant. One important difference, however, is that, apart from maintaining the desired node occupancy, we have to respect the desired values for the BF accuracy.

Let us assume that we have a relation R which is empty and we start inserting values and the corresponding index entries on index key k. The initial node of the BF-Tree is a BF node, as discussed. For each new entry we need to update i) the BF, ii) $\#keys$, iii) possibly min_key or max_key and in some cases iv) the run indexes. When the indexed elements exceed the maximum number of elements that maintains the desired false positive probability fpp we have to perform a node split. Bulk load of an entire index can minimize creation overhead since we can precompute the values of BF-Tree's parameters and allocate the appropriate number of

ALGORITHM 1: Search using a BF-Tree

Search key k using BF-Tree

```
Binary search of root node; read the appropriate child node.
Recursive search the values of the internal node, read the appropriate child node until a leaf is reached.
Read $min\_key$ and $max\_key$ from the leaf node.
if Key $k$ between $min\_key$ and $max\_key$ then
    Probe BF with key $k$.
    if Key $k$ exists in BF then
        Binary search the run indexes.
        Search within the matching data pages for the key (false positive with $fpp$).
    else
        Key $k$ does not exist.
    end if
else
    Key $k$ does not exist.
end if
```

nodes more efficiently. If the tree is in update-intensive mode, each node can maintain a list of inserted/deleted/updated keys (along with their page information) in order to accumulate enough number of such operations to amortize the cost of updating the BF.

ALGORITHM 2: Split a BF-Tree node

Split a BF-Tree node N to N_1, N_2

```
Read $min\_key$ and $max\_key$ of the node to be split.
Create new nodes $N_1$ and $N_2$.
Node split may need to propagate.
for $k = min\_key$ to $max\_key$ do
    if Key $k$ exists then
        if Less than half $N$'s $\in N_1$ then
            Update $min\_key$, $max\_key$, $\#keys$ and $run\_indexes$ of $N_1$ and insert $k$ in $N_1$'s BF.
        else
            Update $min\_key$, $max\_key$, $\#keys$ and $run\_indexes$ of $N_2$ and insert $k$ in $N_2$'s BF.
        end if
    end if
end for
```

The lossy way to keep indexing information for a BF-Tree increases the cost of splitting a BF-leaf. Algorithm 2 shows that in order to split a BF-leaf we need to probe the initial node for all indexed values. Thus, splitting a leaf node is computationally expensive, but it can be accelerated because it is heavily parallelizable. During a node split several threads can probe the BF of the old node in order to create the BF of the new nodes.

Algorithm 3 shows how to perform an insert in a BF-Tree. After navigating towards the BF-leaf in question, we check the BF-leaf size. If this leaf has already indexed the maximum number of values then we perform a node split as described above. After this step, the values of the BF-leaf variables are updated. First, we increase the number of keys ($\#keys$). Second, we may need

ALGORITHM 3: Insert a key in BF-Tree

Insert key k (stored on page p)

```
if $\#keys + 1 \le max\_node\_size$ then
    if $k \notin [min\_key, max\_key]$ then
        Extend range (update $min\_key$ or $max\_key$).
    end if
    Increase $\#keys$.
    Insert $k$ into node's BF.
    Update page list.
else
    Split Node.
    Run insert routine for the newly created node.
end if
```

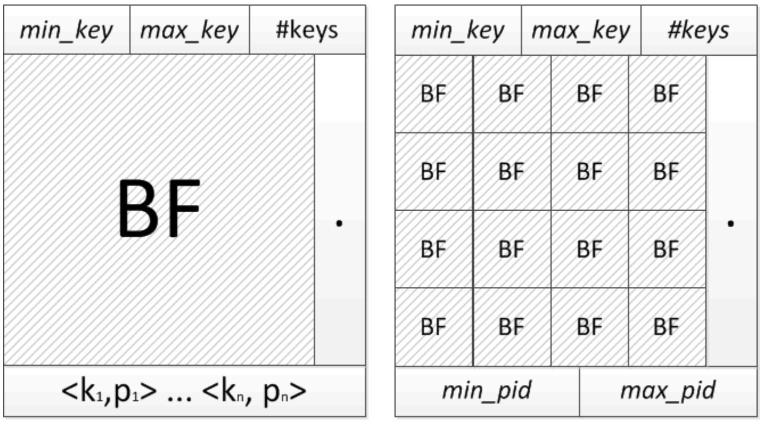
to update the min_key or max_key accordingly. Third, the new key value is added to the BF. Finally, we may have to re-arrange the $run\ indexes$. More precisely, depending on whether the data pages are changed (e.g. a new page is added) we can add a new run-index or re-arrange the run indexes.

Partitioning. A BF-Tree works under the assumption that data are organized (ordered or partitioned) based on the indexing key. We can take advantage of the order of the data if it follows the indexing key, or an implicit order. For example, data like orders of a shop, social status updates or other historical data are usually ordered by date. Thus, any index on the date can use this information. Note, that we do not apply a specific order, we rather simply use the nature of the data for more efficient indexing.

ALGORITHM 4: Search using enhanced BF-Tree

Search key k using enhanced BF-Tree

```
Binary search of root node; read the appropriate child node.
Recursive search the values of the internal node, read the appropriate child node until a leaf is reached.
Read $min\_key$ and $max\_key$ from the leaf node.
if Key $k \in [min\_key, max\_key]$ then
    Probe all BFs with key $k$.
    if Key $k$ exists in at least 1 BF then
        Load the matching pages (false positive with $p$).
    else
        Key $k$ does not exist.
    end if
else
    Key $k$ does not exist.
end if
```



(a) BF-leaf (b) BF-leaf enhanced

Figure 5.4: BF-tree leaves

5.4 Read-optimized BF-Trees

In the previous section, we describe the structure of a BF-Tree, which aims at minimizing the size of the index structure while maintaining similar search time. If the workload consists mostly of analysis and read queries, the indexes can use a pre-processing step which can heavily optimize them for read-accesses. In this section we describe an optimization for BF-Trees which can optimize the search process within a leaf. The core idea of this enhancement is that for each data page that is indexed in a BF-leaf, we create a - much smaller - BF that will index only the keys of the rows that reside in this page. Note that the accuracy of a BF depends on the number of elements inserted. Hence, we can divide the unified BF of the BF-leaf to smaller BFs maintaining the same accuracy. As shown in Section 5.2 (property (1)) dividing a BF for N elements into S BFs for $\frac{N}{S}$ elements each, results in the same false positive probability. This property holds as long as on average each one of the S new BFs will index the same number of elements (i.e., the values are evenly distributed and, thus, every page hold - about - the same amount of rows). For the remainder of the paper the term BF-Tree refers to read-optimized BF-Trees.

BF-leaf enhanced. Consider a BF-leaf in which we index the rows of N data pages. Practically, this BF-leaf holds the meta-data described in Section 5.3, limiting the size of the BF to the remaining space, say B bits. We create N small BFs, which we call page-level BFs (PBF), and each one has B/N bits size (see Figure 5.4(b)). This leads to virtually the same false positive probability with the same aggregate size. An implication of this design is that we need to know the number of data pages corresponding to each leaf. This information however can be easily calculated by the statistics of the data (relation cardinality, histogram of values for the indexed attribute). The BF of a leaf-node is broken down to one page-level BF per data page. The run indexes are not needed anymore. Instead, only the first and the last pages having a PBF in the leaf are needed (two page ids: min_pid and max_pid). For simplicity and consistency we map the first PBF to page min_pid, the second PBF to page min_pid+1 and the last PBF to page

max_pid. Algorithm 4 shows how to perform a search using the read-optimized BF-Tree and Algorithm 5 how to perform an insert.

ALGORITHM 5: Insert a key in an enhanced BF-Tree

Insert key k (stored on page p)

```
if $\#keys + 1 \leq max\_node\_size$ then
    if $k \notin [node\_min, node\_max]$ then
        Extend range (update $node\_min$ or $node\_max$).
    end if
    Increase $\#keys$.
    Insert $k$ into the corresponding BF.
else
    Split Node.
    Run insert routine for the newly created node.
end if
```

Searching in an read-optimized BF-Tree. Performing a search with an read-optimized BF-Tree is based on the search using a simple BF-Tree as far as the internal traversal of the BF-Tree. When the enhanced BF-leaf is retrieved then instead of probing a single BF, every PBF is probed for the search key. Since the data does not have to be ordered and we know that the search key is mapped to the range of the current BF-leaf, every PBF may contain matching tuples. Hence, after probing every PBF we retrieve the pages corresponding to positive matches and we verify whether the search key exists in the pages as with the simple BF-Tree.

Bulk loading BF-Trees. Similarly to other tree indexes, the build time of a BF-Tree can be aggressively minimized using bulk loading. In order to bulk load a BF-Tree the system creates packed BF-leaves with PBF and builds the remaining of the tree on top of the leaves level during a new scan of the leaves. Hence, bulk loading requires one pass over the data and one pass over the leaves of the BF-Tree. Bulk loading applies for the BF-Trees that are not read-optimized as well.

5.5 Modeling BF-Trees and B$^+$-Trees

In this section we present an analytical model to capture the behavior of BF-Trees and compare them with B$^+$-Trees regarding size and performance.

5.5.1 Parameters and model

Table 5.1 presents the key parameters of the model. Most of the parameters are sufficiently explained in the table, however, a number of parameters are further discussed. For a BF-Tree the average occurrence of a value of the indexed attribute ($avgcard$) plays an important role since no new information is stored in the index (effectively reducing its size). Moreover, the desired false positive probability (fpp) allows us to design BF-Trees with variable size and accuracy for exactly the same dataset. Two more parameters characterize BF-Trees: indexed values per leaf ($BFkeysperpage$) which is a function of the fpp and data pages per leaf ($BFpagesleaf$) which is a function of the first and it is related to performance since it is the maximum amount of I/O needed when we want to retrieve a tuple indexed in a BF-leaf. Finally, for the I/O cost there are three parameters, traversing the index (randomly), $idxIO$, random accesses to the data ($dataIO$) and sequential access to the data ($seqDtIO$). That way, we can alter the assumptions of the storage used for the index and the data: either keep them in the same medium (e.g., both on SSD) or store the index on the SSD and the data on HDD.

Table 5.1: Parameters of the model

$Parametername$	$Description$
$pagesize$	pagesize for both data and index
$tuplesize$	(fixed) size of a tuple
$notuples$	size of the relation in tuples
$avgcard$	avg cardinality of each indexed value
$keysize$	size of the indexed value (bytes)
$ptrsize$	size of the pointers (bytes)
$fanout$	fanout of the internal tree nodes
$BPleaves$	number of leaves for the B$^+$-Trees
BPh	height of the B$^+$-Trees
$BPsize$	size of the B$^+$-Trees
fpp	false positive probability for BF-Trees
$BFkeysperpage$	indexed keys per BF leaf
$BFpagesleaf$	data pages per leaf
$BFleaves$	number of leaves for the BF-Tree
BFh	height of the BF-Tree
$BFsize$	size of the BF-Tree
mP	number of matching pages per key
$BPcost$	cost of probing a B$^+$-Tree
$BFcost$	cost of probing a BF-Tree
$BFROcost$	cost of probing an enhanced BF-Tree
$idxIO$	cost of a random traversal of the index
$dataIO$	cost to access randomly data
$seqDtIO$	cost to access sequentially data
$costROFP$	cost of a read-optimized false positive
$seqScan$	cost to seq. access a partition

Equation 5.2 is used to calculate the $fanout$ of the internal nodes of both BF-Trees and B$^+$-Trees. In Equation 5.3 we calculate the number of leaves of a B$^+$-Tree needed based on the data and the indexing details, while the height of the B$^+$-Tree is calculated with the Equation 5.4.

$$fanout = \frac{pagesize}{ptrsize + keysize} \tag{5.2}$$

$$BPleaves = \frac{notuples \cdot (\frac{keysize}{avgcard} + ptrsize)}{pagesize} \tag{5.3}$$

$$BPh = \lceil log_{fanout}(BPleaves) \rceil + 1 \tag{5.4}$$

In order to calculate the leaves of a BF-Tree we first need to calculate the different keys that a BF of a BF-leaf can index (solving in Equation 5.5, Equation 5.1 assuming the bits available in a page) and then plug in this number in Equation 5.6 where we make sure that we correctly calculate the size of the BF-Tree by discarding multiple entries of the same index key. Equation 5.7 calculates the height of the BF-Tree, and Equation 5.8 uses the number of different indexed values per BF-leaf to calculate the the number of data pages per BF-leaf.

$$BFkeysperpage = -pagesize \cdot 8 \cdot \frac{ln^2(2)}{ln(fpp)} \tag{5.5}$$

$$BFleaves = \frac{notuples}{avgcard \cdot BFkeysperpage} \tag{5.6}$$

$$BFh = \lceil log_{fanout}(BFleaves) \rceil + 1 \tag{5.7}$$

$$BFpagesleaf = \frac{BFkeysperpage \cdot avgcard \cdot tuplesize}{pagesize} \tag{5.8}$$

Next, Equations 5.9 and 5.10 estimate the sizes of the trees.

$$BPsize = pagesize \cdot (BPleaves + \frac{BPleaves}{fanout}) \tag{5.9}$$

$$BFsize = pagesize \cdot (BFleaves + \frac{BFleaves}{fanout}) \tag{5.10}$$

Equation 5.11 calculates the average number of pages to be retrieved after a probe with a match (mP). Equation 5.12 calculates the cost of probing a B$^+$-Tree and reading the tuple from its original location. Note that for small $avgcard$ the matching pages are equal to 1. If there is no match, mP is equal to 0. Equation 5.13 calculates the probing cost when using the simple BF-Tree, assuming that on average the number of probes to existing keys is equal to the number of probes to non-existing keys. The term $seqScan$ corresponds to the fact that there should be a sequential scan of the partition in order to find the matching tuples (hence, this happens when there is a match) and the term $fpp \cdot seqScan$ corresponds to the fact that when there is no match, but there is a false positive, the partition should be searched in its entirety as well.

$$mP = \lceil \frac{avgcard \cdot tuplesize}{pagesize} \rceil \tag{5.11}$$

$$BPcost = BPh \cdot idxIO + mP \cdot dataIO \tag{5.12}$$

$$\begin{aligned} BFcost &= BFh \cdot idxIO + seqScan + fpp \cdot seqScan \Rightarrow \\ BFcost &= BFh \cdot idxIO + (1 + fpp) \cdot seqScan \end{aligned} \tag{5.13}$$

Finally, Equation 5.14 calculates the cost of searching with the enhanced BF-Tree. In this equation we need to reintroduce the term mP, which is the number of matching pages when there a positive search on the index. We calculate the cost of a false positive as the cost to retrieve sequentially the false positively attributes pages since all these pages are calculated in search time and will be given to the disk controller as a list of sorted disk accesses.

$$\begin{aligned} BFROcost &= BFh \cdot idxIO + mP \cdot dataIO + fpp \cdot BFpagesleaf \cdot seqDtIO \Rightarrow \\ BFROcost &= BFh \cdot idxIO + mP \cdot dataIO + \frac{8 \cdot avgcard \cdot ln^2(2) \cdot fpp \cdot seqDtIO}{tuplesize \cdot ln(fpp)} \end{aligned} \tag{5.14}$$

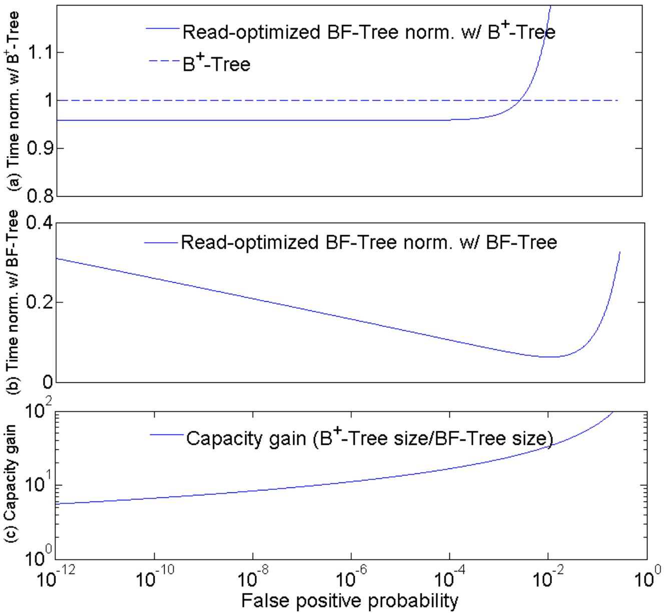


Figure 5.5: Analytical comparison of BF-Tree vs. B$^+$-Tree.

Figure 5.5 presents key comparisons between a B$^+$-Tree and the BF-Trees of a given attribute as an example. In fact, we assume 4KB pages of 256 bytes long tuples with the indexed attribute of size 32 bytes and pointers of size 8 bytes. The relation in this case has size 1GB and the index is stored on SSD and the main data on HDD. The I/O cost is depicted by using the appropriate values of $idxIO$, $dataIO$, and $seqDtIO$. In particular we use $idxIO = 1$, $dataIO = 5$, and $seqDtIO = 2$, modelling an SSD which has random accesses five times faster than random accesses on HDD and two times faster than sequential accesses on HDD. On the x axes of Figures 5.5(a), (b), (c) we vary the desired false positive probability. Figure 5.5(a) shows the read-optimized BF-Tree response time normalized with the B$^+$-Tree response time. We see that BF-Tree can offer better search time for $fpp \leq 0.01$. Figure 5.5(b) shows the read-optimized BF-Tree response time normalized with the simple BF-Tree response time, which shows that in order to have competitive performance we have to use the read-optimized BF-Tree. Hence in Section 5.6 we implement and experiment with the read-optimized BF-Tree. Finally, Figure 5.5(c) shows the capacity gain between the BF-Tree (both variations use the same space) and the B$^+$-Tree, which is corroborated by our experimentation.

5.5.2 Discussion on BF-Tree size and compression

The analysis presented in the previous section regarding the size of the BF-trees and the B$^+$-Trees assumes that the B$^+$-Trees have packed nodes, hence, the comparison takes into account the minimal size of uncompressed dense trees. In this setup, BF-Trees can be up to 10x to 130x smaller than B$^+$-Trees for fpp between 0.0001% and 30%. These numbers are largely corroborated by the experimental results presented in Section 5.6, which shows that approximate indexing offers competitive performance or performance benefits still having significant capacity gains in terms of the index size. In this study we do not consider, however, compression, which is an orthogonal technique to decrease the index size. Compression can decrease the size of both a BF-Tree and a B$^+$-Tree, although, it is more likely to have better results in the B$^+$-Tree since the BFs themselves have relatively small compression ratio.

Several compression techniques have been proposed for B$^+$-Trees: prefix-key compression, delta encoding of keys and/or pointers, and LZ77 compression are typical examples. Aggressively compressing with LZ77 can lead to up to 50% compression rate, however, BF-Trees can achieve one to two order of magnitudes smaller index sizes compared with uncompressed B$^+$-Trees because they use workload knowledge to build the index. Hence, using BF-Tree to minimize index size can outperform the capacity gains of standard index compression techniques.

5.6 Experimental evaluation

We implement a prototype BF-Tree and we compare against a traditional B$^+$-Tree and an in-memory hash index. The BF-Trees are parametrized according to the false positive probability for each BF, which affects the number of leaf nodes needed for indexing an entire relation and, consequently, the height of the tree. The BF-Tree can be built and maintained entirely in main memory or on secondary storage. The size of a BF-Tree is typically one or more orders of magnitude smaller than the size of a B$^+$-Tree, so we examine cases where the BF-Tree is entirely in main memory and cases where the tree is read from secondary storage. We use stand-alone prototype implementations for both B$^+$-Trees and hash indexes. The code-base of the B$^+$-Tree with minor modifications serves as the part of the BF-Tree above the leaves. BF-Trees can be implemented in every DBMS with minimal overhead since they require to add support for BF-leaves, and build their methods as extensions of the typical B$^+$-Tree methods.

5.6.1 Experimental Methodology

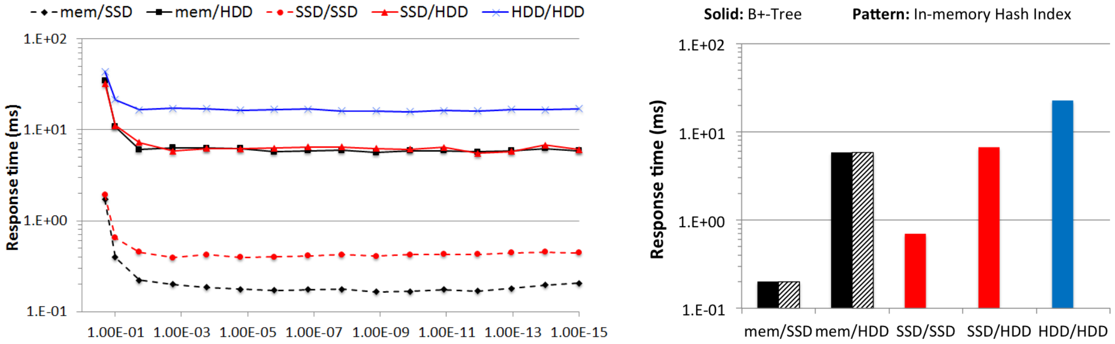
In our experiments we use a server running Red-Hat Linux with 2.6.32 64-bit kernel. The server is equipped with 2 6-Core 2.67GHz Intel Xeon CPU X5650 and 48GB of main memory. Secondary storage for the data and indexes is either a Seagate 10KRPM HDD - offering 106MB/s maximum sequential throughput for 4KB pages - or a OCZ Deneva 2C Series SATA 3.0 SSD - with advertised performance 550MB/s (offering as much as 80kIO/s of random reads). Data on secondary storage (either HDD or SSD) are accessed with the flags O_DIRECT and O_SYNC enabled, hence *without using the file system cache*.

We experiment with a synthetic workload comprised of a single relation R and with TPCH data. For the synthetic workload, the size of each tuple is 256 bytes, the primary (PK) key is 8 bytes and the second attribute we index (ATT1) has size 8 bytes as well, having, however, each value repeated 11 times on average. Both attributes *are ordered* because they are correlated with the creation time. For the TPCH data, we use the three date columns of the lineitem table with scale factor 1. The tuple size is 200 bytes and the indexed attribute is shipdate on which the tuples are ordered. Each date of the shipdate is repeated 2400 times on average. Every experiment is the average of a thousand index searches with a random key. The same set of search keys is used in each different configuration. In the experiment we vary the (i) the false positive probability (fpp) to understand how the BF-Tree is affected by different values and (ii) the indexed attribute in order to show how BF-Tree indexing behaves for a PK and for a sorted attribute. The BFs are created using 3 hash functions, typically enough to have hashing close to ideal. Throughout the experiments the page size is fixed to 4KB. Hence, in order to vary the fpp, the maximum number of keys per BF-leaf is limited accordingly using Equation 5.1.

When the B^+-Tree or the hash index is used during an index probe, the corresponding page is read and, consequently, the tuple in question is retrieved using the tuple id. For the BF-Tree probes, the system reads the BF-leaf which corresponds to the search key and then probes all BFs (one for each data page). The pages for which the answer is positive (potentially, false positive) are sequentially retrieved from the disk and searched for the search key. If it is a primary key search, as soon as the tuple is found the search ends and the tuple is output to the user. If the indexed attribute is not unique then each page is read entirely and if the last tuple has the same search key as the query then the next page is checked as well.

5.6.2 BF-Tree for primary key

We first experiment with indexing the primary key (PK) of R having size 1GB. Each index key exists only once and the data pages are ordered based on it. All index probes in the experiment match a tuple, hence every probe retrieves data from the main file.



(a) BF-Tree performance varying fpp

(b) B^+-Tree/Hash Index performance

Figure 5.6: BF-Tree and B^+-Tree performance for the PK index for five storage configurations for storing the index and the main data.

Build Time and size. The build time of the BF-Tree is one order of magnitude smaller than the build time of the corresponding B^+-Tree following roughly the difference in size between the two trees shown in Table 5.2. The bulk creation of the BF-Tree first creates the leaves of the tree in an efficient sequential manner and then builds on top the remainder of the tree (which is 2-3 orders of magnitude smaller) to navigate towards the desired leaf. The principal goal of the BF-Tree, however, is to minimize the required space. Varying the fpp from 0.2 to 10^{-15} the size of a BF-Tree is 48x to 2.25x time smaller than the corresponding B^+-Tree. Both grow linearly with the number of data pages indexed in terms of the total number of nodes. The capacity gain as a percentage of the size of the corresponding B^+-Tree remains the same for any file size.

Table 5.2: B^+-Tree & BF-Tree size (pages) for 1GB relation

Variation	fpp	Size for PK	Size for ATT1
B^+-Tree		19296	1748
BF-Tree	0.2	406	38
BF-Tree	0.1	578	54
BF-Tree	$1.5 \cdot 10^{-7}$	3928	358
BF-Tree	10^{-15}	8565	786

Table 5.3: False reads/search for the experiments with 1GB data

fpp	False reads for PK	False reads for ATT1
0.2	13.58	701.15
0.1	1.23	80.93
$1.9 * 10^{-2}$	0.11	4.75
$1.8 * 10^{-3}$	0	0.36
$1.72 * 10^{-4}$	0.01	0.04

Response time. Figure 5.6(a) shows on the y-axis the average response time for probing the BF-Tree index of a 1GB relation using the PK index. Every probe reads a page from the main file and returns the corresponding tuple. On the x-axis we vary the fpp from 0.2 to 10^{-15}. As the fpp decreases (from the left-to-right on the graph) the BF-Tree is getting larger but it offers more accurate indexing. The five lines correspond to different storage configurations. The lines with the solid color represent experiments where data are stored on the HDD and the index either in memory (black), on the SSD (red) or on the HDD (blue). The dotted lines show the experiments where data are stored on SSD and the index either in memory (black) or on SSD (red). In order to compare the performance of BF-Tree with that of a B^+-Tree and a hash index we show in Figure 5.6(b) the response time of a B^+-Tree using the same storage configurations and the hash index when index resides in memory. Note that in this experiment the B^+-Tree and every BF-Tree has height equal to 3.

Data on SSD. When the index resides in memory and data reside on SSD BF-Tree manages to match the B^+-Tree performance for $fpp \leq 1.8 \cdot 10^{-3}$, leading to capacity gain 12x. This is connected with the number of falsely read pages per search (see Table 5.3), which are virtually zero for $fpp \leq 1.8 \cdot 10^{-3}$. We observe, as well, a very slow degradation of performance as fpp is getting close to 10^{-15}, which is attributed the larger index size (see Table 5.2); for every positive search we have to read more leaves. We compare the in-memory search time with a hash index which performs similarly to the memory-resident B^+-Tree and hence the optimal BF-Tree. When index is on SSD as well, increased fpp can be tolerated leading to capacity gain 33x with a low performance gain (1.08x), while we can still observe significant capacity gain (12x) with 1.77x lower response time, for $fpp = 0.002$. In the latter case we observe that a higher number of falsely read pages is tolerated because their I/O cost is faster amortized by of the I/O cost to retrieve the index.

Data on HDD. If the index is stored in memory and data on HDD, the PK BF-Tree can still provide significant capacity gains. The BF-Tree matches the B$^+$-Tree for $fpp \leq 1.56 \cdot 10^{-6}$, offering similar indexing performance while requiring 6x less space, but it is already competitive for $fpp = 0.02$ providing capacity gain 19x. The in-memory hash index search shows similar performance to the memory-resident B$^+$-Tree and hence the optimal BF-Tree. If the index is stored on SSD, the BF-Tree outperforms B$^+$-Tree for $fpp \leq 10^{-3}$ offering 12x smaller index size. Finally, if the index is stored on HDD as well then the height of trees dominate performance. Having to read from disk the pages of the two trees (starting from the root) leads to a BF-Tree which can outperform the corresponding B$^+$-Tree even for $fpp = 0.1$ (leading to more than 33x smaller indexing tree). This case however is not likely to be realistic because the nodes of the higher levels of a B$^+$-Tree reside always in memory.

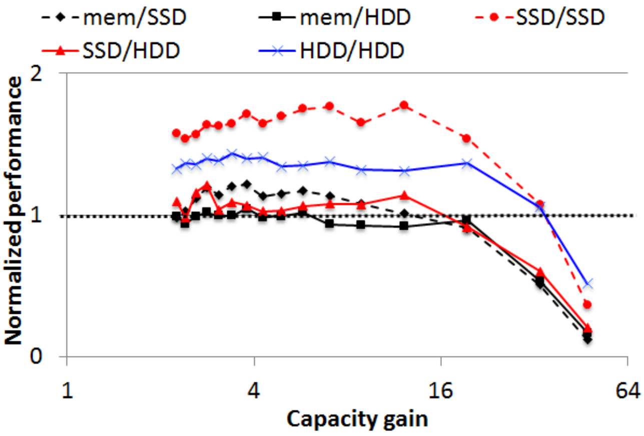
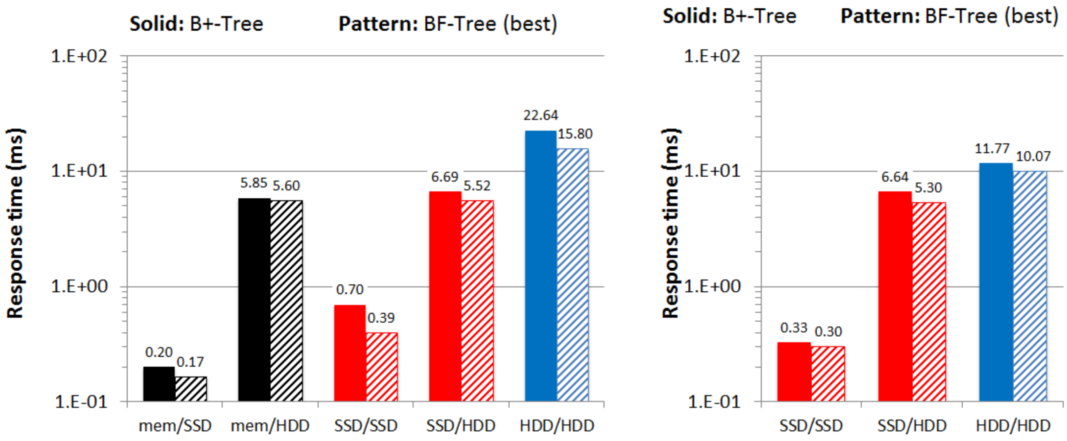


Figure 5.7: The break-even points when indexing PK

Break-even points. A common observation for all storage configurations is that there is a *break-even point* in the size of a BF-Tree for which it performs as fast as a B$^+$-Tree. Figure 5.7 shows on the y-axis the normalized performance of BF-Trees compared with B$^+$-Trees. The x-axis shows the capacity gain: the ratio between the size of the B$^+$-Tree and the size of the BF-Tree for a given fpp. For normalized performance higher than 1, the BF-Tree outperforms the B$^+$-Tree (it has lower response time), for lower than 1 the other way around, and for values equal to 1, the BF-Tree has the same response time as the B$^+$-Tree. The cross-sections between the line with normalized performance equal to 1 and the five lines for the various storage configuration give us the break-even points. As the I/O cost increases (going from memory to SSD or from SSD to HDD) the break-even point shifts towards larger capacity gains, since less accuracy can be tolerated as long as it requires more CPU time (for example probing more BFs, or scanning more tuples in memory) instead of more random reads on the storage medium.

Warm caches. Figure 5.8(a) summarizes the data presented in Figures 5.6(a) and (b) by showing the B$^+$-Tree response time alongside with the fastest BF-Tree response time for an index probe on the PK index. The five sets of bars correspond to the five storage configurations previously described for experiments with cold caches (i.e., one I/O operation for every node that is accessed). In order to see the efficacy of BF-Trees with warm caches, in Figure 5.8(b) we summarize in a similar way the response time of B$^+$-Trees and the best response time of the fastest BF-Tree.

Figure 5.8(b) has only three sets of bars because, trivially, the first two sets of bars would be exactly the same. In the experiments with warm caches, only accessing the leaf node would cause an I/O operation, hence, the height of the tree is not a crucial factor for the response time any more. Moreover, since, typically, the B$^+$-Tree is higher, its performance improvement with warm caches is higher compared to the BF-Tree performance improvement. Comparing the third set of bars of Figure 5.8(a) with the first set of bars of Figure 5.8(b) – both corresponding to the storage configuration when both index and data reside on SSD – we observe that having warm caches results in 2x improvement for B$^+$-Tree and only 25% improvement for BF-Tree, however, still leading to a 10% faster BF-Tree. For the SSD/HDD configuration the improvement is small for both indexes because the bottleneck is now the cost to retrieve the main data. Last but not least, when both index and data reside on HDD, the cost of traversing the index is significantly reduced by almost 2x for B$^+$-Tree and about 33% for BF-Tree, resulting in a 17% faster BF-Tree.



(a) Cold caches (b) Warm caches

Figure 5.8: BF-Tree and B$^+$-Tree performance for PK index: cold caches vs. warm caches.

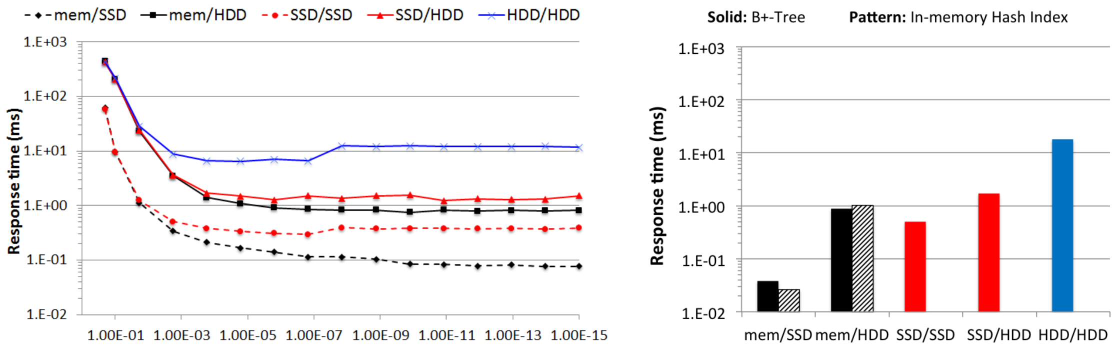
By having warm caches, the fundamental difference is that the height of three plays a smaller part in the response time. Nevertheless, the response time of BF-Tree is lower than the one of B$^+$-Tree in every storage configuration because of the lightweight indexing. The gain in response time depends on the behavior of the storage for the index and for the data. When index and data are stored on the same medium (i.e., both on SSD, or both on HDD) there is small room for improvement, however, when data reside in a slower medium than index, having a lightweight index makes a larger difference.

Partitions of not ordered data with ordered fences. We perform experiments using different data organization. In particular, we shuffle the data respecting, however, the fence of the partition (i.e., the minimum and the maximum value of every BF-leaf). For arbitrary permutations of the keys the behavior of the BF-Tree on average is the same as for the ordered case. This can be explained by the way BF-Tree works. Once we locate the appropriate leaf (i.e., partition), we probe all PBFs to find the correct page. We only read the correct page (and some false positives with the appropriate fpp probability), irrespectively of the order we probe the PBFs.

5.6.3 BF-Tree for non-unique attributes

In the next experiment we index a different attribute which does not have unique values. In particular, in the synthetic relation R the attribute ATT1 is a timestamp attribute. In this experiment 14% of the index probes, on average, have a match.

Build time and Size. The build time of a BF-Tree is one order of magnitude or more lower than the one of a B$^+$-Tree. This is attributed to the difference in size which for attribute ATT1 varies between 46x and 2.22x, for fpp from 0.2 to 10^{-15}, offering similar gains with the PK index.



(a) BF-Tree performance varying fpp

(b) B$^+$-Tree/Hash Index performance

Figure 5.9: BF-Tree and B$^+$-Tree performance for ATT1 index for five storage configurations for storing the index and the main data.

Response time. Figure 5.9(a) shows on the y-axis the average response time for probing the BF-Tree using the index on ATT1, as a function of the fpp. For the B$^+$-Tree every probe with a positive match will read all the consecutive tuples that have the same value as the search key. When the BF-Tree is used for the positive matches (regardless whether they are false positives or actual positive matches) the corresponding page is fetched and every tuple of that page has to be read and checked whether it matches the search key (as long as the key of the current tuple is smaller than the search key). The fpp varies from 0.2 to 10^{-15}. Similarly to the previous experiment, the five lines correspond to different storage configurations. As in Section 5.6.2, the lines with the solid color represent experiments where data are stored on the HDD and the index either in memory (black), on the SSD (red) or on the HDD (blue). The dotted lines show the experiments where data are stored on SSD and the index either in memory (black) or on SSD (red). We compare the performance of BF-Tree with that of a B$^+$-Tree and a hash index (Figure 5.9(b)) using the same storage configurations (the hash index always reside in memory). For this experiment the B$^+$-Tree has 3 levels while, the BF-Trees have 2 levels for $fpp > 1.41 \cdot 10^{-8}$ and 3 levels for $fpp \leq 1.41 \cdot 10^{-8}$ Because of the difference in height amongst the BF-Trees we observe a new trend. For all configurations that the index probe is a big part of the overall response time (i.e., the SSD/SSD and the HDD/HDD) we observe a clear increase in response time when the height of the tree is increased (Figure 5.9(a)). In other configurations this is not the case because the index I/O cost is amortized by the data I/O cost. This behavior exemplifies the trade-off between fpp and performance regarding the height of the tree and is later depicted in Figure 5.10 when the best case for BF-Tree gives 2.8x performance gain for HDD and 1.7x performance gain for SSD while the capacity gain is 5x-7x.

Data on SSD. When the data reside on the SSD and the fpp is high (0.3 to 10^{-3}) there is very small difference between storing the index on SSD or in memory. The reason behind that is that the overhead of reading and discarding false positively retrieved pages dominates the response time. Thus, in order to match the response time of the B^+-Tree when index is on SSD the fpp has to be $2 \cdot 10^{-3}$, giving capacity gain 12x, while the in memory BF-Tree never matches the in-memory B^+-Tree nor the hash index.

Data on HDD. The increased number of false positives per index probe (Table 5.3) has increased impact when data are stored on HDD. Every false positively retrieved page incurs an additional randomly located disk I/O. As a result, in order to see any performance benefits the false positively read pages have to be minimal. When the index resides in memory, the BF-Tree outperforms the B^+-Tree and the hash index for $fpp \leq 2 \cdot 10^{-6}$. The break-even point when the index resides on the SSD is shifted for higher fpp ($2 \cdot 10^{-3}$) because a small number of unnecessary page reads can be tolerated as they are being amortized by the cost of accessing the index pages on the SSD. This phenomenon has bigger impact when the index is stored on the HDD. The break-even point is now further shifted and for capacity gain 12x there is already a performance gain of 2x.

Break-even points. Similarly to the PK index, the ATT1 BF-Tree indexes have break-even points, shown in Figure 5.10. Qualitatively, the behavior of the BF-Tree for the five different storage configurations is similar but the break-even points are now shifted towards smaller capacity gains. The reason being mainly the increased number of false positively read pages. In order for the BF-Tree to be competitive it needs to minimize the false positives. On the other hand, the impact of increasing the false positive (and thus, reducing the tree height) is shown by the sudden increase in performance gain particularly for the HDD/HDD (blue line) and the SSD/SSD (dotted red line).

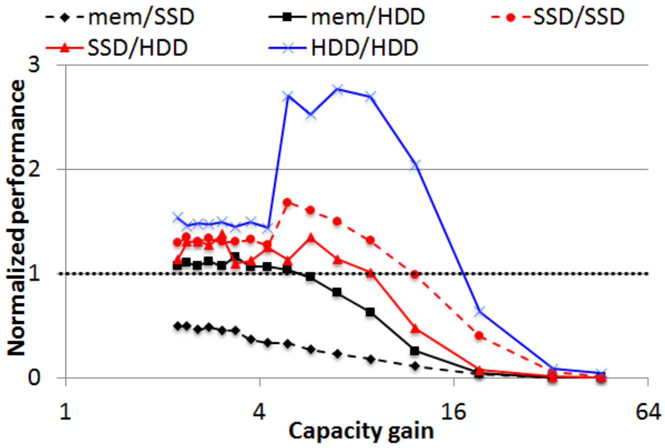
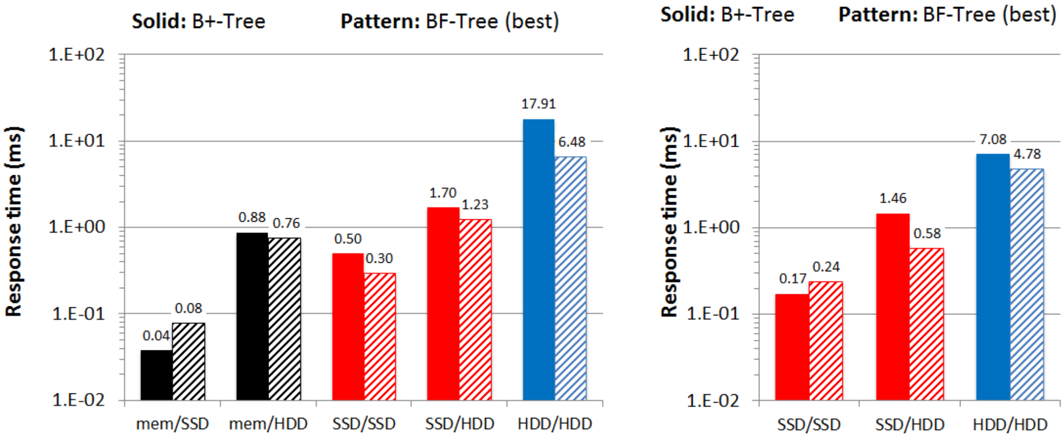


Figure 5.10: The break-even points when indexing ATT1

Benefits for HDD. Our experimentation shows that using BF-Trees when data reside on HDD can lead to higher capacity gains before performance starts decreasing. In fact this is not possible with the simple BF-Trees (Section 5.3) and it is made possible because we use the read optimized BF-Trees (Section 5.4). The symmetric way to see this, is that with the same capacity requirements using BF-Trees on top of HDD gives higher performance gains (in terms of normalized performance). This is largely a result of the enhancement algorithm of BF-Trees which minimizes the occurrence of random reads, the main weak point of HDD.

Warm caches. Similarly to the PK experiments, we repeat the ATT1 experiments and we present the results with warm caches. Figure 5.11(a) summarizes the results of Figures 5.9(a) and (b), while Figure 5.11(b) presents the results with warm caches. As in the PK index case, the improvement for the B$^+$-Tree is higher than the improvement for BF-Tree. In addition, for the storage configuration that both index and data reside on SSD, B$^+$-Tree is, in fact, faster than BF-Tree by 30% because the overhead of the false positives overthrow the marginal benefits of the lightweight indexing. For the other two configurations, however, BF-Tree is faster (2.5x for SSD/HDD and 1.5x for HDD/HDD) because any additional work is hidden by the cost of retrieving the main data which now reside on HDD.

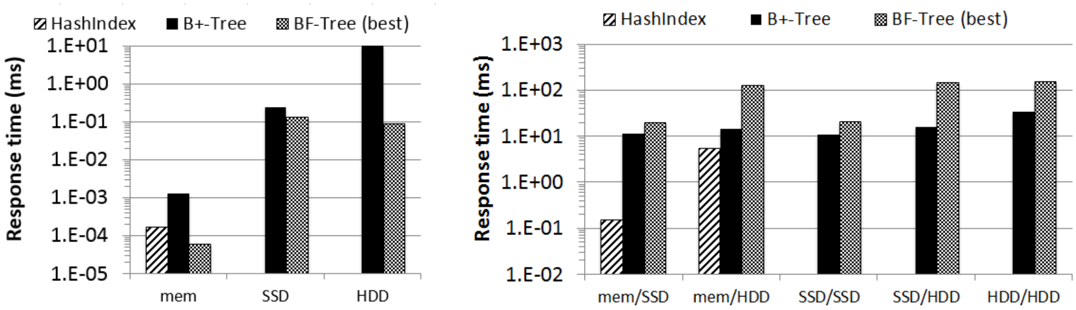


(a) Cold caches (b) Warm caches

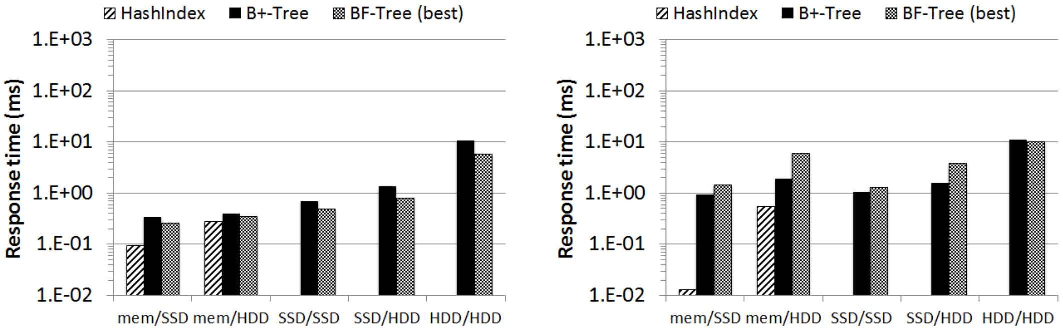
Figure 5.11: BF-Tree and B$^+$-Tree performance for ATT1 index: cold caches vs. warm caches.

5.6.4 BF-Tree for TPCH

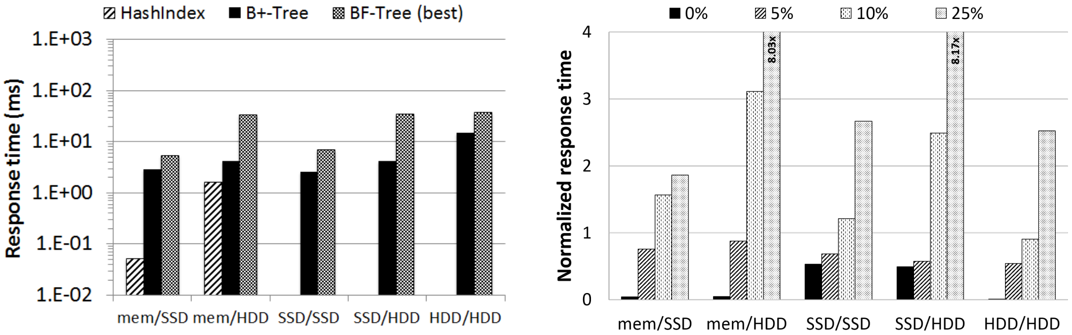
Figures 5.12(a) thought (e) show the response time of a hash index, a B$^+$-Tree and the optimal BF-Tree for the TPCH table lineitem for index probes on the shipdate attribute (on which it is partitioned). Similarly to the five configurations previously described, Figures 5.12(b) though (e) have five sets of columns, while Figure 5.12(a) has three because data are never read (hit rate is 0%) - and the impact of false positives is negligible. In these figure the y-axes show response time in a logarithmic scale. BF-Tree performance for different fpp has very low variation, because of the fact that the high cardinality of each date results in short trees. We observe, however, large differences in the behavior of BF-Trees vs. the one of B$^+$-Trees for different hit rates. Figure 5.12(a) shows that when all index probes end up requesting data that do not exist, BF-Tree is faster than both hash index and B$^+$-Tree when the index is in memory. When the index is kept



(a) TPCH 0% hits (b) TPCH 100% hits



(c) TPCH 5% hits (d) TPCH 10% hits



(e) TPCH 25% hits (f) Normalized response time

Figure 5.12: BF-Tree for point queries on TPCH dates varying hit rate.

in secondary storage a similar behavior is observed. This effect is pronounced for HDD where each additional level of the tree is adding a high overhead. On the other hand in Figure 5.12(b) we see that when the hit rate is 100% the BF-Tree cannot keep up with neither the hash index for the in-memory setup nor the B$^+$-Tree. When the index is on SSD, however, the performance penalty is lower since the overhead of additional random reads on SSD is smaller than what it is on HDD.

Figures 5.12(c), (d), and (e) depict the response times for other values of hit rate: 5%, 10% and 25% respectively. As the hit rate increases the response time of the BF-Tree increases and for 10% or more BF-Tree response time is higher than B$^+$-Tree response time. Figure 5.12(f) shows the same data from a different perspective, by plotting on the y-axis the normalized response time (BF-Tree response time normalized to B$^+$-Tree response time) for the five storage configurations, varying the hit rate from 0% to 25%. When the bars are lower than one BF-Tree is faster which happens for every configuration for hit rate 0% or 5%. One additional observation is that if both data and index are stored in the same medium (i.e., both on SSD or both on HDD) the performance penalty is smaller. Hence, for hit rate 10%, using the configuration SSD/SSD the best BF-Tree is 20% slower than the B^+-Tree and using the configuration SSD/SSD the best BF-Tree is 10% faster than the B^+-Tree (instead of 1.6 to 3x slower which is the case for the other three configurations). The explanation is that for these two configurations the cost of traversing the B^+-Tree dominates the overall response time and BF-Tree is competitive because of its shorter height. In the above experiments the BF-Trees were 1.6x-4x smaller than the B^+-Trees.

5.6.5 Summary

In this section we compare BF-Trees with B^+-Trees and hash indexes in order to understand whether BF-Trees can be both space and performance competitive. We show that depending on the indexed data and the storage configuration there is a BF-Tree design that can be competitive with B^+-Trees both in terms of size and response time. Moreover, for the in-memory index configurations, BF-Trees are competitive against hash indexes as well. The number of false positively read pages has an impact in each and every one of the five storage configurations. When data reside on SSD the number of falsely read pages affect the performance only when it dominates the response time. Random reads do not hurt SSD performance, hence, the BF-Tree performance drops gradually with the number of false positives. When data are stored on HDD the performance gains are high for no false positives but drop drastically as soon as unnecessary reads are introduced.

5.7 BF-Tree as a general index

5.7.1 BF-Tree vs. interpolation search

The datasets used to experimentally evaluate BF-Tree are either ordered or partitioned. For the case that data are ordered a good alternative candidate would be to use binary search or interpolation search (Perl et al., 1978). Interpolation search can be very effective for canonical datasets achieving $log(log(N))$ search time, in the specific case that the values are sorted and evenly distributed.[4] B^+-Trees performance serves as a more general upper bound since binary search average response time is $log_2(N)$ and B^+-Trees average response time is $log_k(N)$, where N is the size of the dataset and the k is the number of <*key, pointer*> pairs a B^+-Tree page can hold. In addition, B^+-Trees serve as a baseline for comparing the size of an index structure used to enhance search performance.

BF-Trees can be used as a general index which is further discussed in this section. Data do not need to be entirely ordered (being partitioned is enough), hence, BF-Tree is a more general access

---

[4] A more widely applicable version of interpolation search has also been discussed (Graefe, 2006a).

method than interpolation search.

5.7.2 Range scans

In addition to the evaluation with point queries here we discuss how BF-Tree support range scans. In fact, we show that BF-Trees or read-optimized BF-Trees have competitive performance for range scans.

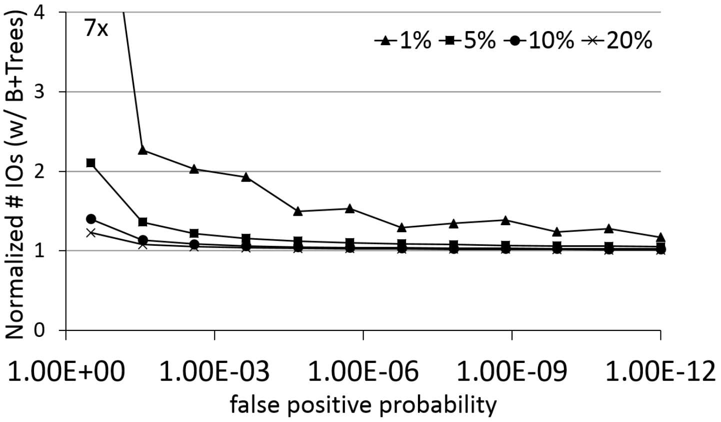


Figure 5.13: IOs for range scans using BF-Trees vs. B^+-Trees.

The BF-leaf corresponds to one partition of the main data. When a range scan spanning multiple BF-Tree partitions is evaluated, the partitions are either entirely part of the range, being *middle partitions*, or part of the boundaries of the range, being *boundary partitions*. The boundary partitions, typically, are not part of the range in their entirety. In this case, the range scan shows a read overhead. An optimization - only possible for the read-optimized BF-Trees - is to enumerate the values corresponding to the boundary partitions and probe the PBF in order to read only the useful pages. Note that within the partition the pages do not need to be ordered on the key. Such an optimization, however, is not practical when the values have a theoretically indefinite domain, or even a domain with very high cardinality. Figure 5.13 shows the number of I/O operations on the main data when executing a range scan using a BF-Tree, normalized with the number of I/O operations needed when a B^+-Tree is used. In the x-axis the fpp is varied from 0.3 to 10^{-12}. The four lines correspond to different ranges, varying from 1% to 20%. We use the synthetic dataset described in Section 5.6, and the indexed attribute is the primary key. A key observation is that as the fpp decreases the partitions hold less values, hence, the overhead of reading the boundary partitions decreases. Hence, we observe that for $fpp \leq 10^{-4}$ for ranges larger than 5% there is negligible overhead. In the case of 1% range scan with $fpp \leq 10^{-6}$, the overhead is less than 20%, and for $fpp \leq 10^{-12}$ the overhead is negligible for every size of the range scan.

5.7.3 Querying in the presence of inserts and deletes

Both inserts and deletes affect the fpp of a BF. In particular, if every BF is allowed to store additional entries for the values falling into their range the fpp is going to gradually increase. If we assume M bits for the BF, and a given initial fpp, then using Equation 5.1, we calculate that such a BF indexes up to $N = -M \cdot ln^2(2)/ln(fpp)$ elements. If we allow to index more elements, i.e., increase N, the effective fpp will in turn increase following a near linear trend for small changes of N (note that the new_N is the number of indexed elements after a number of inserts, hence $new\_N = N + inserts$):

$$new\_fpp = e^{\frac{-M \cdot ln^2(2)}{new\_N}}, \quad \text{and because,}$$

$$-M \cdot ln^2(2) = N \cdot ln(fpp)$$

we have,

$$new\_fpp = e^{\frac{N}{new\_N} ln(fpp)} = fpp^{\frac{N}{new\_N}} = fpp^{\frac{N}{N+inserts}} = fpp^{\frac{1}{1+\frac{inserts}{N}}} \Rightarrow$$

$$new\_fpp = fpp^{\frac{1}{1+insert\_ratio}} \tag{5.15}$$

The new_fpp depends on the number of inserts. Equation 5.15 does not depend on the BF size, nor on the number of elements. It depends on the initial fpp and on the relative increase of indexed elements ($insert\_ratio = \frac{inserts}{N}$).

The behavior of a BF in the presence of inserts is depicted in Figure 5.14. The x-axis is the insert ratio, i.e., the number of inserts as a percentage of the initially indexed elements and the y-axis is the resulting fpp. The are three lines corresponding to initial fpp equal to 0.01% (black line), 0.1% (red dotted line), and 1% (blue dashed line) respectively. In the x-axis the insert ratio varies between 0 and 60x, meaning that the inserted elements were 60 times more than the initial elements. The new_fpp for every initial fpp initially increases super-linearly and for insert ratio more than 15x it continues to increase sub-linearly. The most interesting part of the graph, however, is the initial part that the insert ratio is a small percentage of the initial elements of the BF.

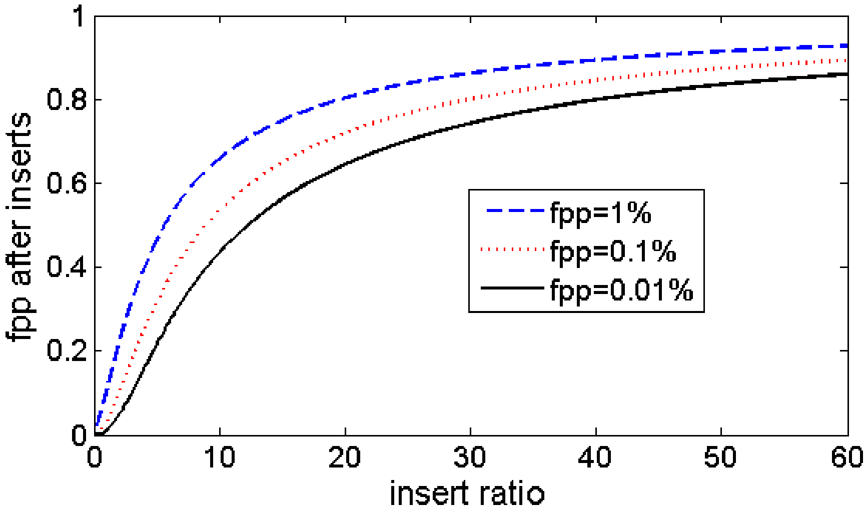


Figure 5.14: False positive probability (fpp) in the presence of inserts.

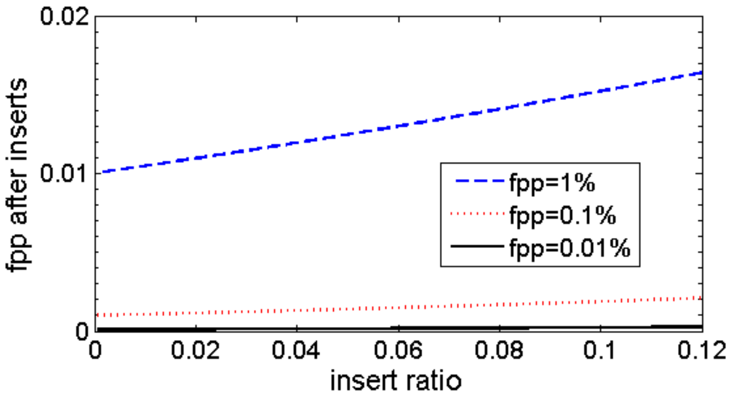


Figure 5.15: False positive probability (fpp) in the presence of inserts (zoomed in).

Hence, in Figure 5.15 we zoom in the x-axis for insert ratio 0 to 12%. We observe that all three lines have a liner trend, and more importantly, the line corresponding to 0.01% has a small increase even for 12% increase of the indexed elements. For example, starting from fpp = 0.01%, for 1% more elements, $new\_fpp \approx 0.011\%$, and for 10% more elements, $new\_fpp \approx 0.23\%$.

Similarly, for deletes the fpp increases because we artificially introduce more false positives. In fact, the number of deletes affects directly the fpp. If we remove 10% of the entries, $new\_fpp = fpp + 10\%$. The above analysis assumes no space overhead. On the other hand, we can maintain the desired fpp by splitting the nodes when the maximum tolerable fpp is reached. That way inserts will not affect querying accuracy (as described in Section 5.3). We can maintain a list of deleted keys in order to avoid increasing the fpp, which are used to recalculate the BF from the beginning when such a list has reached the maximum size. A different approach is to exploit variations of BFs that support deletes (Bender et al., 2012; Rothenberg et al., 2010) after considering their space and performance characteristics.

5.8 Optimizations

Exploiting available parallelism. With the current approach the BFs of a leaf node are probed sequentially. A BF-leaf may index several hundreds data pages leading to an equal number of BFs to probe. These probes can be parallelized if there are enough CPU resources available. In the conducted experiments we do not see a bottleneck in BF probing, however, for different applications, BF-Trees may exhibit such a behavior.

Complex index operations with BF-Trees. In this chapter, we cover how the index building and the index probing are performed and we briefly discuss how updates and deletes are handled. Traditional indexes, however, offer additional functionality. BF-Trees support *index scans* similarly to a range scan. Since data are partitioned, with one index probe we find the starting point of the scan, and then a sequential scan is performed. *Index intersections* are also possible. In fact, the false positive probability for any key after the intersection of two indexes will be the product of the two probabilities (for each index), and hence, typically much smaller than any of the two.

5.9 Conclusions

In this chapter, we make a case for approximate tree indexing as a viable competitor of traditional tree indexing. We propose the Bloom filter tree (BF-Tree) which is a tree structure able to index a relation with sorted or partitioned attributes. BF-Trees parametrize the accuracy of the indexing as a function of the size of the tree. Thus, B^+-Trees are the extreme where accuracy and size are maximum. BF-Trees allow to decrease the accuracy and, hence, the size of the index structure by introducing (i) a small number of unnecessary reads and (ii) extra work to locate the desired tuple in a data page. We show, however, through both an analytical model and experimentation with a prototype implementation, that the introduced overhead can be amortized, hidden, or even superseded by reducing the I/O accesses when a desired tuple is retrieved. Secondary storage is moving from HDD - with slow random accesses and cheap capacity - to the diametrically opposite SSD. BF-Trees achieve competitive performance and minimize index size for SSD, and even when data reside on HDD, BF-Trees index it efficiently as well.

6 Graph Data Management using Solid-State Storage[1]

6.1 Introduction

Solid State Storage (a.k.a. Storage Class Memory (Freitas, 2009)) is here to stay. Today, a well known Solid State Storage technology is NAND flash. Another technology on the horizon is Phase Change Memory (PCM). Both can be used in chip form, for example, as a small storage element in a portable device. The read and write latencies of PCM cells are very low, already in the same ballpark as DRAM (Chen et al., 2011; Doller, 2009). For large scale storage, many chips can be packaged into a single device that provides the same functionality as disk drives, supporting the same basic APIs. SSDs can provide much faster random I/O than magnetic disks because there is no mechanical latency between requests. We focus here on database applications that demand enterprise level storage in this form factor.

NAND flash technology is relatively mature and represents the state-of-the art in the marketplace. Companies have been building storage devices out of flash chips for two decades, and one can find a huge variety of flash-based devices from consumer to enterprise storage. PCM is a relative newcomer, and until now there has been little opportunity to evaluate the performance characteristics of large scale PCM devices. In this chapter we provide insights on where solid state devices can offer a big advantage over traditional storage and highlight possible differences between two representative technologies, flash and PCM.

As discussed earlier, flash devices have superior random read performance compared to magnetic hard-drives but suffer from several limitations. First, there is a significant asymmetry in read and write performance. Second, only a limited number of updates can be applied on a flash device before it becomes unusable; this number is decreasing with newer generations of flash (Abraham, 2010). Third, writing on flash not only is much slower than reading and destructive of the device, but it has proven to interfere with the redirection software layers, known as Flash Translation Layers (FTL) (Bouganim et al., 2009).

---

[1] The material of this chapter has been the basis for the ADMS 2012 "Path Processing using Solid State Storage" (Athanassoulis et al., 2012).

PCM addresses some of these challenges. The endurance of PCM cells is significantly higher than NAND flash (Ovonyx, 2007), although still not close to that of DRAM. Unlike NAND flash, PCM does not require the bulk erasure of large memory units before it can be rewritten. Moreover, while cost is still uncertain, for our purposes, we assume normal cell size competitiveness and standard volume economics will apply to this technology as it ramps into high volume.

The most pronounced benefit of solid state storage over hard disks is the difference in response time for random accesses. Hence, we identify *dependent reads* as an access pattern that has the potential for significant performance gains. Latency-bound applications like path processing (Gubichev and Neumann, 2011) in the context of graph processing, or RDF-data processing are typical examples of applications with such access patterns. The Resource Description Framework (RDF) (RDf, 2013) data model is widely adopted for several on-line, scientific or knowledge-based datasets because of its simplicity in modelling and the variety of information it can represent.

We find that PCM-based storage is an important step towards better latency guarantees with no bandwidth penalties and we identify a trade-off between maximizing bandwidth and minimizing latency. In order to measure the headroom of performance benefit (decrease of response time) in long path queries we implement a simple benchmark and we compare the response times when using flash and PCM. We observe that PCM can yield 1.5x to 2.5x smaller response times for any bandwidth utilization without any graph-aware optimizations[2] some of which we leave for future work. We take this observation one step further and we design a new data layout suitable for RDF data and optimized for a solid state storage layer. The proposed layout increases the locality of related information and decreases the cost of graph traversals by storing more linkage information (i.e., metadata about how to navigate faster when a graph is traversed).

6.1.1 Contributions

We have the following main contributions:

- We devise a custom benchmark to highlight the qualitative and quantitative differences between two representative solid state devices (flash and PCM).
- We find that PCM can natively support higher workload parallelization with near-zero latency penalty — an observation that can be used to shift the algorithm design.
- We find that applications with *dependent reads* are natural candidates for exploiting PCM-based devices. Our graph-based benchmark allows us to measure the benefit that path traversal queries can have from such devices.

---

[2] Such optimizations, in addition to the new data layout presented in this chapter, include complementary pages comprised of auxiliary data structures such as: (i) cached attributes, (ii) aggregate links, (iii) node popularity, (iv) node priority and (v) reverse links.

- We develop RDF-tuple, a new data layout for RDF data which is compatible with traditional DBMS.
- We use the RDF-tuple data layout and we develop a prototype RDF repository to illustrate a specific application that can benefit from PCM storage. Our prototype corroborates the opportunity analysis performed using the graph-based benchmark.

Our PCM device is a prototype, with a device driver having limited sophistication. It is possible that the commercial version of this PCM device could perform better when brought to market.

6.1.2 Outline

The rest of the chapter is organized as follows. Section 6.2 presents the custom graph benchmark and identifies potential performance improvements when using PCM in path processing. Section 6.3 presents the RDF repository and shows that it can achieve the anticipated improvements. Section 6.4 discusses further opportunities and limitations that PCM presents. Section 6.5 includes the details of the queries used in our experiments, and we conclude in Section 6.6.

6.2 Path Processing

Current PCM prototypes behave as a "better" flash (Jung et al., 2011), in the sense that they have faster and more stable reads. We argue that the best way to make the most of this behavior is in the domain of applications with *dependent reads*. Hence, we create a simple benchmark that performs path traversals over randomly created graphs to showcase the potential benefits of using PCM as secondary storage for such applications. (More information about performance characteristics of the devices we used can be found in Chapter 2, Section 2.1.5).

Custom graph-based benchmark. We create a benchmark that we call the *Custom graph-based benchmark*. We model path traversal queries by graph traversal over a custom built graph. The graph (see description in Table 6.1) is stored in fixed-size pages (each page has one node) and the total size of the graph is 5GB. Each node has an arbitrary number of edges (between 3 and 30). The path traversal queries are implemented as link following traversals of a random edge in each step. Each query starts from a randomly selected node of the graph and it follows at random one of the descendant nodes. When multiple queries are executed concurrently, because of the absence of buffering, locality will not yield any performance benefits. Each query keeps reading a descendant node as long as the maximum length is not reached.

Table 6.1: Description of the benchmark

Dataset	Randomly generated graph
Degree	Randomly between 3 and 30
# nodes	1.3M (approximately)
Size on disk	5GB

Experimental setup. We use a 74GB FusionIO ioDrive (SLC) (FusionIO, 2010) and a 12GB Micron PCM prototype (SLC). The PCM device offers 800MB/s maximum read bandwidth, 20μs hardware read latency[3] (for 4K reads) and 250μs hardware write latency (for 4K writes), while the endurance is estimated to be 10^6 write cycles. While PCM chips can be byte-addressable the PCI-based PCM prototype available to us uses 4KB pages for reads and writes[4].

Table 6.2: Hardware specification of the PCM device

Brand	Micron
PCM type	Single Level Cell (SLC)
Integration	90nm
Size	12GB
Interface	PCI-Express 1.1x8
Read Bandwidth	800MB/s (random 4K)
Write Bandwidth	40MB/s (random 4K)
H/W Read Latency	20μs (4K)
H/W+S/W Read Latency	36μs (4K)
H/W Write Latency	250μs (4K)
H/W+S/W Write Latency	386μs (4K)
Endurance	10^6 write cycles per cell

The flash device offers 700MB/s read bandwidth (for 16K accesses) and hardware read latency as low as 50μs in the best case. The details about the two devices can be found in Tables 6.2 and 6.3. The system used for the experiments was a 24-core 2.67GHz Intel Xeon X5650 server with the 74GB ioDrive and the 12GB PCM device. The operating system is Ubuntu Linux with the 2.6.32-28-generic x86_64 kernel and the total size of available RAM is 32GB.

Table 6.3: Hardware specification of the flash device

Brand	FusionIO
NAND type	Single Level Cell (SLC)
Integration	30-39nm
Size	74GB
Interface	PCI-Express x4
Read Bandwidth	700MB/s (random 16K)
Write Bandwidth	550MB/s (random 16K)
Mixed Bandwidth	370MB/s (70R/30W random 4K mix)
IOPS	88,000 (70R/30W random 4K mix)
H/W Read Latency	50μs Read (512b)
H/W+S/W Read Latency	72μs (4K)
H/W+S/W Write Latency	241μs (4K)
Endurance	24yrs (@ 5TB write-erase/day)

Experimental evaluation. We present a set of experiments based on the custom graph benchmark. We compare flash and PCM technology as secondary storage for path queries. We vary the page size (and consequently node size), the length of the

Table 6.4: Custom graph benchmark parameters.

Path length	2, 4, 10, 100 nodes per query
Concurrent threads	1, 2, 4, 8, 16, 32, 64, 96, 128, 192
Page Size	4K, 8K, 16K, 32K
Page processing time	0μs, 50μs, 100μs

---

[3] Software read latency is about 16–17μs, which is negligible compared to magnetic disk I/O latency, but is close to 50% of the total latency for technologies like PCM.

[4] It is noteworthy that, unless the s/w stack is optimized, smaller page accesses will make the impact of s/w latency a bigger issue as the percentage of s/w latency over the total page read latency will increase (Akel et al., 2011).

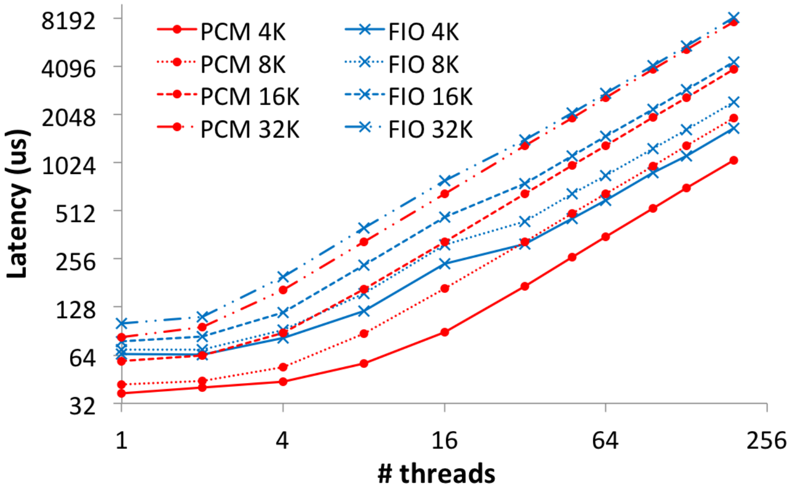


Figure 6.1: Latency for 4K, 8K, 16K, 32K page size when varying concurrent queries. Note the logarithmic scale on both axes.

path, the number of concurrent requests and the page processing time, all of them summarized in Table 6.4. The variation of the values for each parameter plays a different role: different path lengths help us reveal if there is a cumulative effect when a query over a longer path is evaluated; different numbers of concurrent threads show the impact of multiple I/O requests on flash and PCM; different page sizes (a.k.a. access granularity) signify which is the best mode of operation for each device; and varying page processing time helps us understand what is the benefit when the workload is "less" I/O-bound. We observe that varying the path length leads to the same speedup because the reduced latency remains the same throughout the execution of the query regardless of the length of the query. Moreover, higher page processing time reduces gradually the benefit of PCM, which is expected since the longer IO time for the flash device is amortized over the page processing time. In particular, when page processing is increased from $0\mu s$ to $100\mu s$ the maximum speedup that PCM offers is reduced by 20%.

Figure 6.1 presents the average latency and Figure 6.2 the average bandwidth during the execution of the custom graph benchmark, when we vary the page size and the number of concurrent threads. The page processing time for the presented experiments is set to $0\mu s$ and the path length to 100 nodes.

Figure 6.1 shows the average read latency per I/O request (y-axis) as a function of the number of concurrent threads issuing queries (x-axis) for different page sizes (different graphs). The red lines correspond to PCM and the blue lines to flash. In all four cases (page size equal to 4K, 8K, 16K, 32K) PCM shows lower I/O latency but the best case is when the page size is smallest, i.e., 4K. Extrapolating this pattern we anticipate that when PCM devices have smaller access granularity (512-byte pages are part of the roadmap) the benefit will become even larger.

In Figure 6.2 we see the bandwidth achieved when running the custom graph benchmark. Simi-

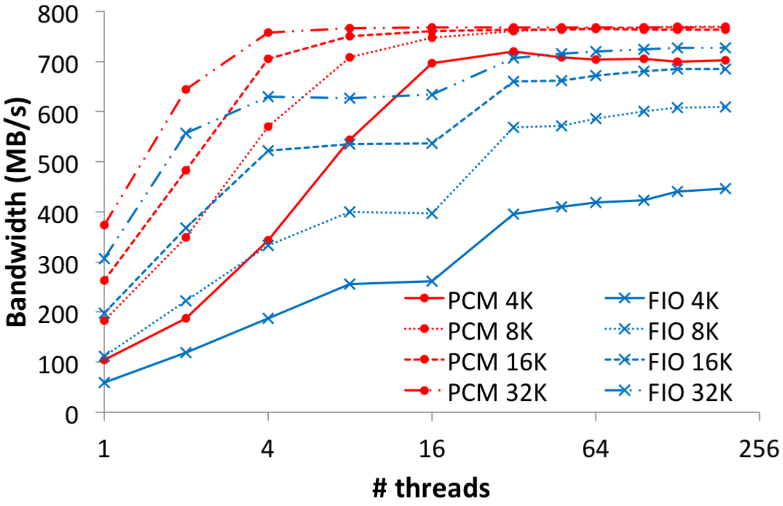


Figure 6.2: Bandwidth for 4K, 8K, 16K, 32K page size when varying concurrent queries

larly to the previous graphs, the x-axis of the different graphs represents the number of threads issuing concurrent requests, the y-axis is bandwidth and different graphs correspond to different page sizes. The capacity for both devices is roughly 800 MB/s. The PCM device reaches very close to the maximum sustained throughput (close to 800MB/s) with 4K pages when 16 queries are issued concurrently. When the page size is increased the maximum can be achieved when 4-8 queries are issued concurrently. When increasing the page size and the number of concurrent queries any additional (small) benefit has a high cost in increased latency. On the other hand, flash has qualitatively different behavior. For each page size the maximal bandwidth is achieved with 32 concurrent queries and a bigger page size is needed for a better result. Using the *iostat* and *sar* Linux tools we were able to verify that when 32 queries are issued concurrently, the full potential of the flash device can be achieved by increasing the I/O queue size to 64 (which is the maximum possible) without having delays due to queuing. Thus, having 32 concurrent queries is considered to be the sweet spot where flash has the optimal bandwidth utilization. Going back to the latency figures we can now explain the small bump for 16 threads. In fact, the observed behavior is not a bump at 16 threads but an optimal behavior at 32 threads. This phenomenon is highlighted in Figure 6.3.

In Figure 6.3 we present the speedup of the query response time for different values of path length and number of concurrently issued queries. The speedup varies between 1.5x and 2.5x having the maximum number of 16 threads. We observe as well that the length of the query does not play any important role. The sudden drop in speedup for 32 threads is attributed to the previously described sweet spot for flash for this number of concurrent queries. When we increased page processing time from $0\mu s$ to $100\mu s$ the maximum speedup was reduced from 2.5x to 2.0x. Figure 6.4 shows a different way to read the data from Figure 6.1. On the x-axis we vary the page size (4K, 8K, 16K, 32K) and on the y-axis we present the latency per I/O when we run the custom graph benchmark. The solid bars correspond to experiments using 4 threads and the

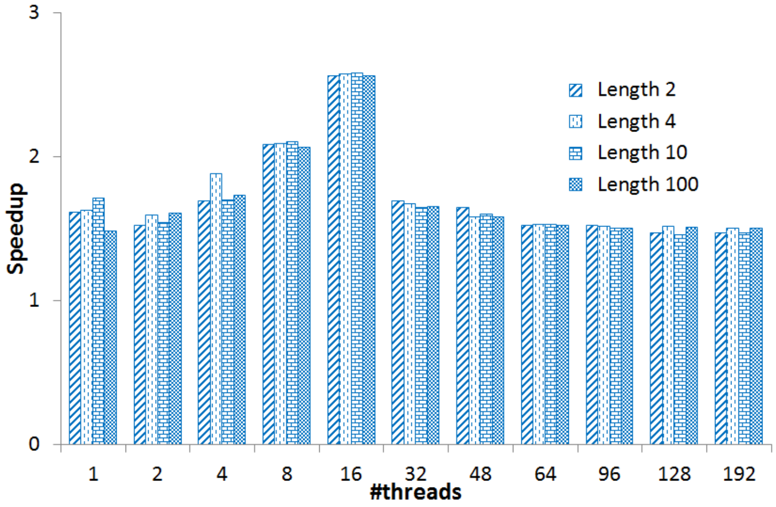


Figure 6.3: Custom path processing benchmark: Speedup of PCM over flash

checkered bars to experiments using 16 threads, the red color represents PCM and the blue color flash. There are several messages to be taken from this graph. First, when page size is equal to 4K (the best setting for both devices) using PCM leads to the highest benefit in latency (more than 2x speedup). Secondly, we observe that the average latency per I/O when using PCM with 16 threads is almost the same (7% higher) compared to average latency per I/O when using flash with 4 threads. In other words we can have 4 times more queries accessing 4 times more data with the similar latency per group of I/O (which corresponds to 4 reads for flash and 16 reads for PCM). This observation can be used to create search algorithms for PCM which can take advantage of *near-zero penalty concurrent reads*, which we outline in Section 6.4. Similarly, for 8 concurrent threads reading 4K pages from PCM the latency is $57\mu s$ and for 2 concurrent threads reading 4K pages from flash the latency is $65\mu s$ allowing PCM to fetch 4 times more data in less time. Finally, we see that the benefits from PCM decrease as we increase page size, which help us make the case that we should expect even higher benefits when PCM devices offer finer access granularity.

Figure 6.5 presents the bandwidth as a function of page size. Even for 4K page size, PCM can achieve 22% higher bandwidth with 4 threads than the flash device with 16 threads. If we compare the 16-thread cases, we can almost saturate PCM with 4K page size. On the other hand, we are unable to saturate the flash device even with 32K page size. The last two figures demonstrate that a PCM device can show important latency and bandwidth benefits, relative to flash, for workloads with dependent reads.

6.3 RDF processing on PCM

In this section we describe our prototype RDF repository, called *Pythia*. We identify the need to design an RDF-processing system which takes into account the graph-structure of the data and

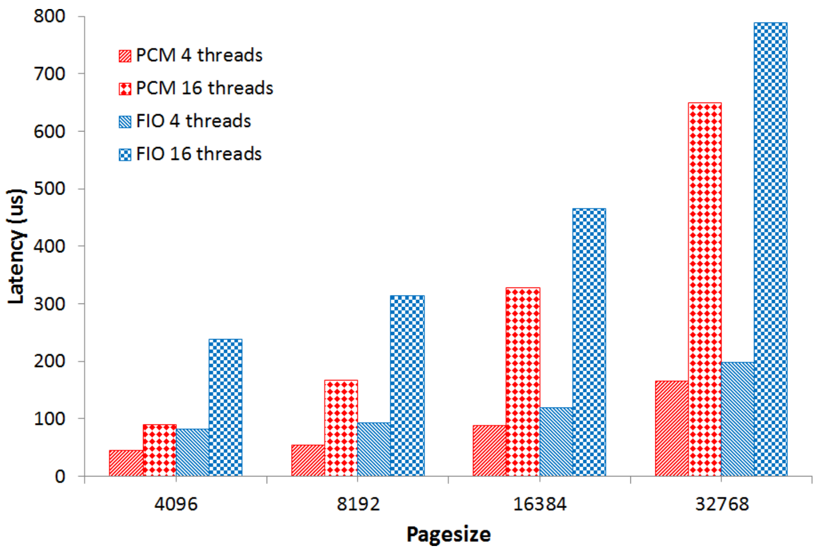


Figure 6.4: Latency per page request size for PCM and flash, varying page size and number of concurrent threads issuing requests.

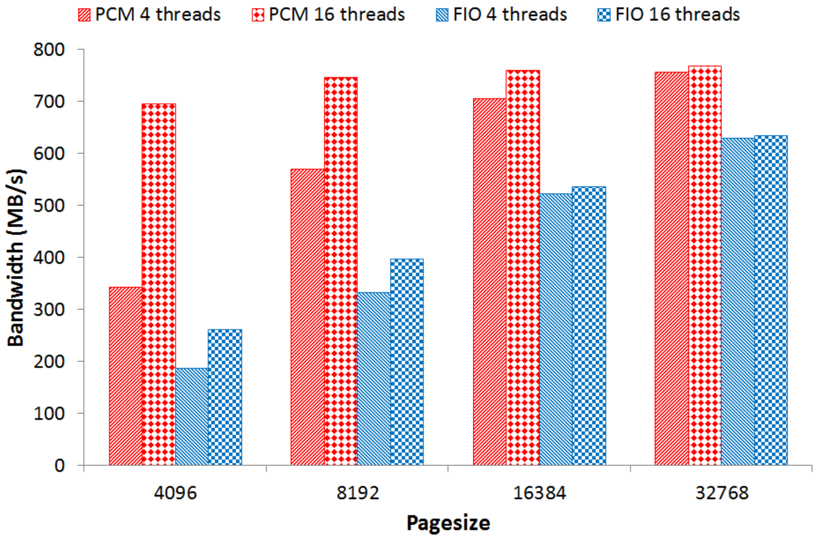


Figure 6.5: Bandwidth for PCM and flash, varying page size and number of concurrent threads issuing requests.

has the infrastructure needed to support any query over RDF data. Pythia is based on the notion of an *RDF-tuple*.

6.3.1 The RDF-tuple

An *RDF-tuple* is a hierarchical tree-like representation of a set of triples given an ordering of subject, predicate, and object. In Pythia, we will store RDF-tuples for two complementary hierarchies: the subject/predicate/object (SPO) hierarchy and the object/predicate/subject (OPS) hierarchy. Each triple will thus be represented in two places, the SPO-store and the OPS-store.

In the SPO store, the root of the tree contains the information of the common subject of all triples. The children of the subject-node are the property-nodes, one for each property that is connected with the given subject. For each property-node, its children are in turn the identifiers of object-nodes that are connected with the subject of the root node through the property of its parent property-node. The RDF-tuple design allows us to locate within a single page[5] the most significant information for any given subject. Furthermore, it reduces redundancy by omitting repeated instances of the subject and predicate resources. Conceptually, the transformation of RDF triples to an RDF-tuple in the SPO-store is depicted in Figure 6.6. The OPS-store is analogous.

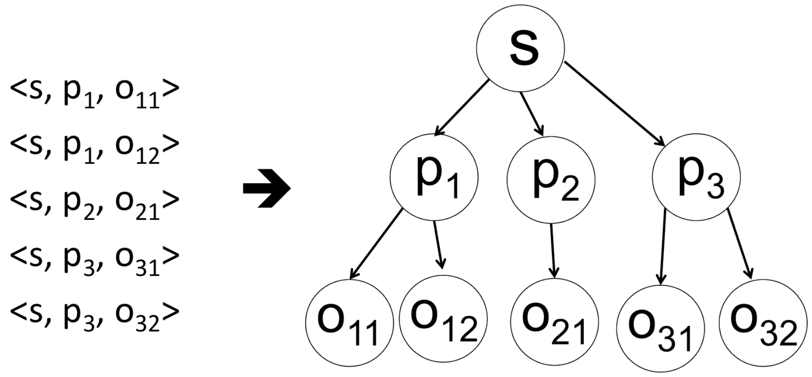


Figure 6.6: RDF-tuple in the SPO-store

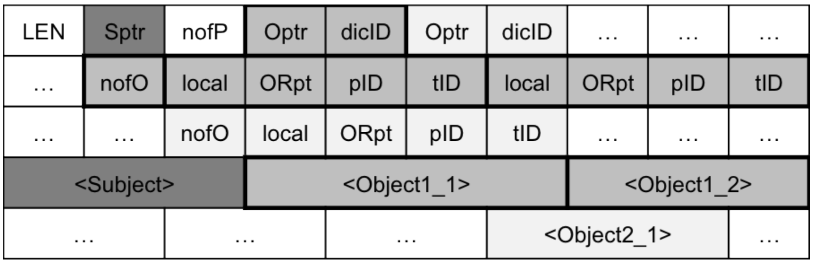
We chose to materialize the SPO and OPS hierarchy orders in Pythia, but not other orders such as PSO. This choice is motivated by our observation that the large majority of use cases in the benchmarks of Section 2.6.1 need subject-to-object or object-to-subject traversal. Usually the query had a subject (or object) in hand and needed to find connections of given types to other objects (subjects). Rarely did the query specify a predicate and ask for all object/subject pairs. We thus avoid the extra storage requirements of representing additional versions of the data, at the cost of occasionally needing a more expensive plan for a predicate-oriented query.

We envision RDF-tuples to be stored as a tuple with variable length in a database page containing

---

[5] See Section 6.3.2 for a discussion of tuples that don't fit in a page.

many such tuples. Figure 6.7 shows how an RDF-tuple is laid out within a page.



LEN	Sptr	nofP	Optr	dicID	Optr	dicID	…	…	…
…	nofO	local	ORpt	pID	tID	local	ORpt	pID	tID
…	…	nofO	local	ORpt	pID	tID	…	…	…
<Subject>			<Object1_1>				<Object1_2>		
…		…		…		<Object2_1>			…

Figure 6.7: RDF-tuple layout

We employ a standard slotted page scheme with variable size tuples, but the internal tuple organization is different from prior work. An RDF-tuple is organized in two parts: the metadata part (first three lines of Figure 6.7) and the resource part (the remaining lines). In the metadata part the above tree structure is stored with internal tuple pointers (offset) to the representation of the nodes and the resources. The offsets to the resources point to the appropriate locations in the resource part of the tuple. In more detail, the metadata part consists of the following variables:

- the length of the tuple,
- the offset of the subject's resource,
- the number of predicates,
- for each predicate the offset of the predicates resource and an offset to the objects (for the combination of subject and predicate),
- for each predicate's objects: the number of objects, the object's resource offset, a flag saying whether the resource is stored locally and the page id and tuple id for every object as a subject.

In this representation there are several possible optimizations. For example, the resource of the predicate can typically be stored in a dictionary because in every dataset we analyzed the number of different predicates is very small (in the order of hundreds). Moreover, in the case that the object is a literal value (e.g., a string constant) the value can be stored in a separate table with a unique identifier.

6.3.2 The Internals of Pythia

The SPO-store and the OPS-store are each enhanced by a hash index; on the Subject for the SPO store and on the Object for the OPS store. There is a separate repository for "very large objects" (VLOBs), i.e., RDF-tuples which require more storage than what is available in a single page. VLOBs are important when the dataset includes objects that are connected with many subjects

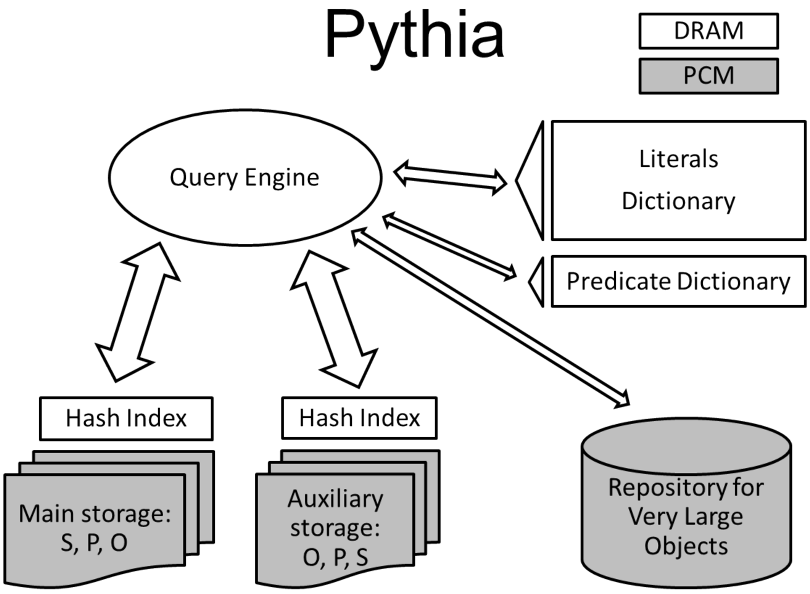


Figure 6.8: Pythia Architecture

and create a huge RDF-tuple for the OPS store. (Imagine the object “human”; a variety of other subjects will be linked with this using the predicate “is-subclass-of”, leading to a huge OPS RDF-tuple.) Additionally, we employ two dictionaries mainly for compression of the RDF-tuples: the *Literals dictionary* and the *Predicate dictionary*. The former is used when a triple consists of a literal value as an object and the latter is used to avoid storing the resource of the predicates multiple times in the two stores. In the majority of RDF datasets the variability of predicates is relatively low leading to a small Predicate dictionary, always small enough to fit in memory. On the other hand, different datasets may lead to a huge Literal dictionary and may need a more complex solution.

Figure 6.8 presents the architecture of the prototype RDF repository Pythia we designed and experimented with. The grayed parts (SPO-store, OPS-store and VLOB-store) reside in secondary storage. The remaining parts are stored in persistent secondary storage, but are maintained in RAM during the time the system is running.

6.3.3 Experimental Workload

We use the popular dataset Yago2 (Hoffart et al., 2011) to investigate the benefits of the proposed approach in an environment with *dependent reads*. Yago2 is a semantic knowledge base derived mainly from Wikipedia, WordNet and GeoNames. Using the RDF-tuple design for the SPO- and OPS-stores, we store the initial 2.3GB of raw data (corresponding to 10M entries and 460M facts) into a 1.5GB uncompressed database (the SPO-store and OPS-store together account for 1.3GB, and VLOBs account for 192MB). The large objects can be aggressively shrunk down by employing page-level compression, an optimization which we leave for future work. When the system operates the in-memory structures require 121MB of RAM for the hash indexes and 569MB of RAM for the dictionaries, almost all of it for the Literals dictionary. Another future optimization for the literals is to take into account the type and store them in a type-friendly compressible way. In the SPO store more than 99% of the RDF-tuples fit within a 4K page.

We hard-code and experiment with six queries over the Yago2 dataset:

1. Find all citizens of a *country* that have a specific *gender*.
2. Find all events of a specific *type* that a given *country* participated in.
3. Find all movies of a specific *type* that a given *actor* participated in.
4. Find all places of a specific *type* that are located in a given *country* (or in a place ultimately located in this *country*).
5. Find all ancestors of a given *person*.
6. Find all ancestors with a specific *gender* of a given *person*.

The equivalent SPARQL code for these queries can be found in Section 6.5.

The first three queries search for information of the given subject or object; in other words they are searching for *local knowledge*. The last three queries require a traversal the graph in an iterative mode to compute the correct answer. The last three queries typically require dependent reads.

6.3.4 Experimental Evaluation

We used the same 24-core Intel Xeon X5650 described previously. Similarly to the previous experiments, we experimented using as back-end storage a state-of-the-art 80GB FusionIO flash device (SLC) and a 12GB Micron PCM device prototype. The workload for the first two experiments was a mix of the six aforementioned queries with randomized parameters. We executed the workload using either flash or PCM as main storage in anticipation of a benefit similar to what our custom graph benchmark showed in Section 4.

Figure 6.9 shows the throughput achieved by Pythia when Flash (red bars) or PCM (blue bars) is used for secondary storage for a varying number of threads submitting queries concurrently. The black line shows the relative speedup when PCM is employed.

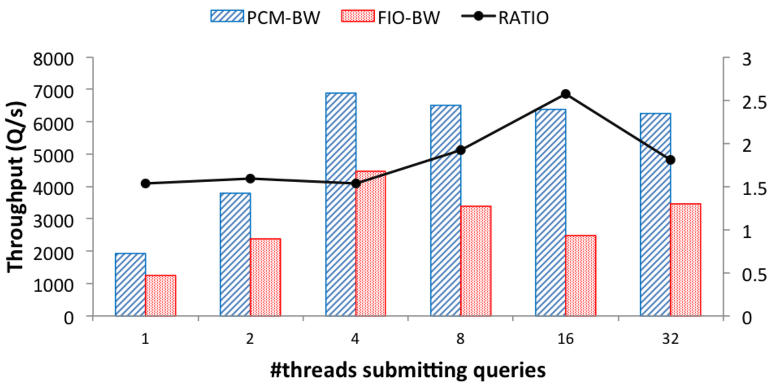


Figure 6.9: Bandwidth for Pythia for 1, 2, 4, 8, 16, 32 concurrent clients submitting queries

In this experiment we corroborate the performance benefit achieved in the simple benchmark with a more realistic system and dataset. Both devices scale close to linearly until 4 threads (7KQ/s for PCM and 4.5KQ/s for flash leading to 1.56x speedup), while for higher number the PCM device is more stable showing speedup from 1.8x to 2.6x.

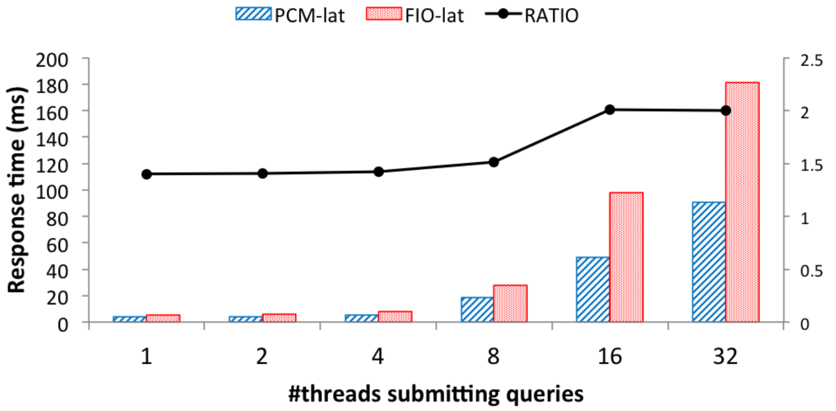


Figure 6.10: Latency for Pythia for 1, 2, 4, 8, 16, 32 concurrent clients submitting queries

Figure 6.10 shows the response time achieved by Pythia when Flash (red bars) or PCM (blue bars) is used for secondary storage for a varying number of threads submitting queries concurrently. The black line shows the relative speedup when PCM is employed. In terms of response time PCM is uniformly faster than flash from 1.5x to 2.0x.

6.3.5 Comparison with RDF-3X

In this section we compare Pythia with RDF-3X (Neumann and Weikum, 2008) showing that our approach is competitive against the research prototype considered to be the current state-of-the-art. We instrument the source code of RDF-3X with time measurement code and we time the pure execution time of the query engine of RDF-3X without taking into account the parsing or

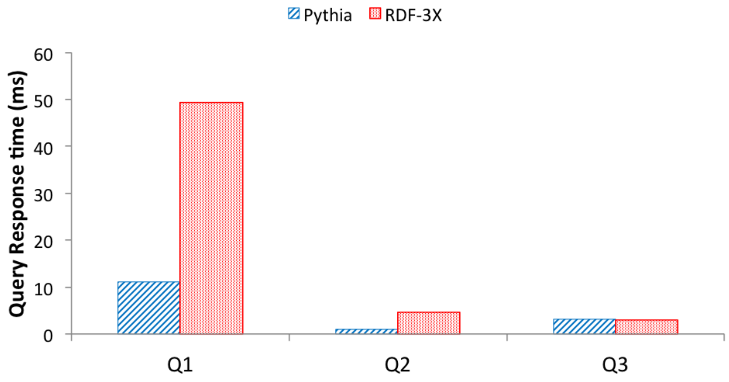


Figure 6.11: Query latency using Pythia for Q1, Q2 and Q3

optimization phases. We compare the response time of three queries when we execute them using Pythia and RDF-3X. We store the Yago2 data on PCM and we use the following three queries to understand whether Pythia is competitive with an established RDF repository.

Q1. Find all male citizens of Greece.

Q2. Find all OECD member economies that Switzerland deals with.

Q3. Find all mafia films that Al Pacino acted in.

Figure 6.11 shows that Pythia (blue bars) performs equally well or faster than RDF-3X (red bars) for all three queries. For Q1 Pythia is 4.45x faster, which can be attributed to the fact that the gender information for every person is stored in the same page as the rest of the person's information, thus incurring no further IOs to read it. In Q2 the benefit of Pythia is about 4.69x for the same reason: the information about whether a country is an OECD member is stored in the same page with the country. The third query incurs a higher number of IOs in both cases because we have to touch both the SPO- and OPS-stores, hence Pythia and RDF-3X perform almost the same (RDF-3X is 3% faster).

6.4 PCM Opportunities

Any device that can handle concurrent requests in parallel can be kept busy if there are enough independent I/O streams. Under such conditions, the device will saturate its bandwidth capabilities. When there are too few independent I/O streams to saturate the device with a single I/O request per stream, one might allow each I/O stream to submit multiple concurrent requests.

When the access pattern requires dependent reads, identifying concurrent I/O units for a single stream can be challenging. Roh et al. (Roh et al., 2011) consider a B^+-tree search when multiple individual keys are being searched at the same time. If multiple child nodes of a single internal node need to be searched, I/Os for the child nodes can be submitted concurrently. This is a dependent-read pattern in which multiple subsequent reads may be dependent on a single initial read. Roh et al. demonstrate performance improvements for this workload on several flash devices. They implemented an I/O interface that allows the submission of several I/O requests with a single system call, to reduce software and hardware overheads.

Both PCM devices and flash devices can handle a limited number of concurrent requests with a minor latency penalty. For our PCM device running with 8 concurrent threads, we can achieve roughly double the bandwidth and half the latency of the flash device, without significantly increasing the latency beyond what is observed for single-threaded access. Thus our PCM device has the potential to outperform flash when I/O streams can submit concurrent I/O requests.

6.4.1 Algorithm redesign

A graph-processing example of such a workload is *best-first search* (BestFS). In best-first search, the priority of nodes for subsequent exploration is determined by an application-dependent heuristic, and nodes at the current exploration fringe are visited in priority order. Breadth-first search (BFS) and depth-first search (DFS) are special cases in which the priority function is suitably chosen. On a device that efficiently supports n concurrent I/O requests, we can submit the top n items from the priority queue as an I/O batch, effectively parallelizing the search. The children of the newly fetched nodes are inserted into the priority queue for the next round of the search algorithm.

In particular, BestFS, implements a hardware optimized BFS with DFS performance guarantees. The main concepts of a BestFS are (i) the I/O parallelism of the device and (ii) the heuristic to be used. The execution of BestFS proceeds as follows. The search begins from a given node of the graph, and all the neighboring nodes are imported in the ToExamine queue, where they are ordered based on the heuristic used. Then, n nodes are serviced in parallel. The number n is dictated by the I/O parallelism of the device. When, n nodes are services in parallel they are fetched by storage in the same time as a single node, and then the addresses of their descendants are imported in the ToExamine queue, when all the nodes are re-ordered based on the heuristic used. This process continues iteratively, until we find the node we are looking for. It is worth mentioning that, a BestFS execution can reach the furthest descendant in the same time as DFS if the execution is I/O bound because every batch of reads happens in the same time as a single read. In this time, however, the BestFS is capable of testing more than the nodes consisting of the path to the furthest descendant having performed, in fact, a partial BFS.

6.5 SPARQL code for test queries

In this section we present the equivalent SPARQL version of the queries implemented during the evaluation of Pythia in Section 6.3. These queries are used to compare against the research prototype RDF-3X (Neumann and Weikum, 2008) using the *pathfilter* operator introduced in the literature (Gubichev and Neumann, 2011).

1. Find all citizens of a *country* that have a specific *gender*.
   SPARQL: select ?s where { ?s <isCitizeonOf> *country*. ?s <hasGender> *gender* }
2. Find all *type* events that a *country* participated in.
   SPARQL: select ?s where { *country* <participatedIn> ?s. ?s <type> *eventType* }
3. Find all *type* movies that an *actor* participated in.
   SPARQL: select ?s where { ?s <type> *movieType*. *actor* <actedIn> ?s }
4. Find all places of a specific *type* that are located in a *country* (or in a place ultimately located in this *country*).
   SPARQL: select ?s where { ?s <type> *placeType*. ?s ?path *country*. pathfilter(containsOnly(??path, <isLocatedIn>)) }
5. Find all ancestors of a *person*.
   SPARQL: select ?s where { ?s ??path *person*.
   pathfilter(containsOnly(??path, <hasChild>)) }
6. Find all ancestors having gender *gender* of a *person*.
   SPARQL: select ?s where { ?s ??path *person*. ?s <hasGender> *gender*. pathfilter(containsOnly(??path, <hasChild>)) }

Q1. Find all *male* citizens of *Greece*.
select ?s where { ?s <hasGender> <male>. ?s <isCitizenOf> <Greece> }

Q2. Find all *OECD member economies* that *Switzerland* deals with.
SPARQL: select ?s where
{ ?s <type> *wikicategory_OECD_member_economies*.
<Switzerland> <dealsWith> ?s }

Q3. Find all *mafia* films that *Al Pacino* acted in.
SPARQL: select ?s where { ?s <type> *wikicategory_Mafia_films*. <Al_Pacino> <actedIn> ?s }

6.6 Conclusions

In this chapter we present a new physical data layout for RDF data motivated by the rapid evolution of the available storage technologies. We use as our secondary storage two representative solid-state storage devices: a state-of-the-art flash device (fusionIO) and a real-life PCM-based (Micron) prototype. The proposed data layout called RDF-tuple is suitable for general RDF queries and is implemented and tested in a prototype storage engine called Pythia, showing competitive performance when compared with the research state-of-the-art system RDF-3X. RDF-tuple can natively store RDF information in a DBMS friendly way and exploit both existing DB algorithms and custom graph-based algorithms.

Our experiments with solid-state storage show that graph processing can take advantage of it naturally because they employ *dependent reads*. Commercial scale PCM technology is relatively new, but we see that it is already competitive against mature flash devices. A key observation is that a PCM device with a very basic device driver can outperform a mature flash rival in terms of the read latency and read bandwidth of a single device. Our experimentation with a custom graph benchmark shows that using an early prototype PCM device can yield significant performance benefits over flash when running path traversal queries (1.5x-2.5x).

7 The Big Picture

In this thesis we set the stage for efficient scaleup data analytics by building hardware-aware and workload-aware system design. We integrate non-volatile solid-state storage in the data management system hierarchy and we build algorithms that use workload characteristics to offer better performance and increased functionality. We study different datasets and workloads taken from, or similar, to typical data analytics applications.

The reason to choose this research direction stems from the increasing importance of data analytics in modern applications, including business, research, governmental and scientific applications. The means of collecting and storing information evolves, creating new challenges in supporting modern analytics applications. Hence, the question of how to store and how to analyze data is of paramount importance. At the same time, the underlying hardware is evolving and new storage and processing technologies become available allowing for radical system and algorithm redesign.

In this chapter we summarize the contributions of this thesis and we discuss a number of interesting research topics related with the current hardware trends and data analytics.

7.1 What we did

In this thesis we build algorithms and tools and we design systems that use solid-state storage in order to address the requirements of modern data analytics: fast query response times ensuring data freshness, support for increasing concurrency, efficient storage, indexing and accessing of both archival and knowledge-based datasets. We demonstrate the importance of exploiting new storage technologies in concert with algorithms tailored for data analytics applications. By exploiting analytical query patterns and solid-state storage we offer efficient online updates with near-zero overhead in query response times. For workloads with increased concurrency, we show that work sharing allows us to use less resources offering better throughput and very small latency penalty. Moreover, we demonstrate that using the data organization we can offer space efficient and competitive indexing. Finally, we show that in order to integrate knowledge-based data, a new data-layout offering both locality for the properties of each subject, and fast linkage retrieval, provides faster response times than the state-of-the-art.

This thesis is a crucial step towards modern data analytics, studying the impact of both hardware-aware optimizations and workload-aware design. In this thesis we discuss the importance of the storage trends on the way we store, query and analyze data, while at the same time we show the performance and functionality benefits when pre-existing workload knowledge is taken into account for the system design. In the center of the proposed work is the understanding that the constant evolution of storage creates the need to study and reconsider database system design since the characteristics of secondary storage are weaved with the traditional DBMS design. To that extent, we discuss in this chapter new research directions that lie ahead stemming from the changes in storage technologies yet to come. In addition, we discuss one fundamental trend in data analytics which will drive the design of future engine: concurrency in workloads.

7.2 Storage is constantly evolving: Opportunities and challenges

The past decade we have witnessed a tremendous shift in the database systems design fueled by the emergence of multicore architectures and deeper memory hierarchies as a direct consequence of the increasing gap between processor and memory speeds. Today we observe an equally important technological shift in the storage layer, which motivated part of the work presented in this thesis. Flash devices are already used in enterprise deployments but the optimal way to exploit such devices depends on the application. In fact, flash and other solid-state devices are used in a wide range of ways, varying from an intermediate caching layer (following the trend of deeper storage hierarchies) to stand-alone storage. Additionally, new technologies - with Phase Change Memory and Memristor on the edge of being commercially available - create new opportunities and new challenges for research on database system design.

In fact solid-state devices based on flash technology are bound to face two challenges: the capacity and endurance walls. As the integration granularity of CMOS chips has become finer it is increasingly difficult to pack more flash capacity (i.e., floating gates) in the same surface. Companies building storage devices project that there are a few more generations of increasing the storage density for flash. In order to delay the *capacity wall*, flash device vendors shifted from single-level cell (SLC) devices to multi-level cell (MLC) devices which store more than one bit per storage cell. This strategy is causing the *endurance wall* because erase blocks of MLC devices can sustain 2-3 orders of magnitude less erasures than the ones of SLC devices before they become unreadable. Designing systems that use flash devices for persistent storage in the presence of this constantly shifting tradeoff between endurance and capacity is an open research topic.

7.2.1 Persistent main memory: Database Management Systems vs. Operating Systems

Storage technologies that are still under research and development - like Phase Change Memory - are expected to offer fine-grained addressability and non-volatility. In order to design a system with persistent main memory, however, these two properties are not enough. Similarly to the consistency and durability properties of database transactions, programs that operate on top of persistent main memory need to have data structures that can ensure consistency of the persistent data. In some cases durability may not be necessary, but when durability is pursued there is need for a mechanism for consistency. First, in the micro-architectural level current systems do not guarantee the order that writes in memory are performed nor whether a write that returned to the calling function is completed. Second, unless the response time of accessing a chip of a future persistent main memory is in the same order as today's main memory it will be impossible to exploit the fine-grained addressability because the cost of accessing the persistent memory has to be amortized. Addressing the research problems of persistent main memories require research on the micro-architectural level and on the application level in tandem.

7.2.2 Software stack is too slow

The recent advancements in storage technologies have revealed a very important characteristic of the software stack sitting on top of the storage devices. The software layers of operating system and file system have been designed having in mind response times in the order of milliseconds (or more) from the main storage devices (Akel et al., 2011; Caulfield et al., 2012). Today, however, solid-state storage devices have one or more orders of magnitude faster response times. Production flash devices have response times in the order of 50μs and Phase Change Memory prototypes in the order to 20μs (Athanassoulis et al., 2012), making the latency from the software layers (the *software latency*) comparable with the access latency for reading from or writing to the device (the *hardware latency*). Trimming down the software layer can be achieved, for example, by exposing directly the device to the application, or by building new lightweight file systems.

7.3 Ever increasing concurrency in analytics

The nature of analytical queries is expanding from offline analysis to ubiquitous queries. For example, asking from our smartphone the current consumption of minutes and data of our monthly plan initiates a personalized analytical query. Moreover, in recent research it is documented that customers of data analytics products indicate that they will have to routinely execute several hundreds of concurrent queries in the near future (Candea et al., 2011). A report in trends in data warehousing published in 2012 (Russom, 2012) anticipates at least one order of magnitude increase in concurrent users submitting queries in data analytics systems over the next few years. Hence, rethinking the query-centric approach and building query engines tailored for high concurrency is key to deliver the data analytics engines of the future.

7.3.1 Extending support of work sharing

A first step towards data analytics engines supporting high concurrency is the recent introduction of work sharing approaches. A detailed comparison of two such approaches is presented in this thesis. This analysis paves the way for more query engines to integrate work sharing techniques in order to exploit both planned and ad hoc sharing. Work sharing in production systems needs a more sophisticated multi-query optimization method and meticulous optimization depending on the hardware and software platform.

7.3.2 Column-stores and high concurrency

Column-store systems have been designed in order to support analytical workloads, however, ever increasing concurrency is creating new challenges. A study of the behavior of three popular column-store systems when running analytical queries with increasing concurrency (Alagiannis et al., 2013) shows that the tested systems cannot sustain stable performance, after they become saturated, which is attributed to the lack of aggressive dynamic resource reallocation. For increasing concurrency the throughput is decreased drastically but the available processors are not always fully utilized. Inefficiencies of the query engines and the lack of aggressive dynamic resource reallocation or of CPU-aware admission control leads to queries underutilizing available resources. This behavior leaves room for further research aiming at efficiently supporting high concurrency in column-stores. Work sharing is no stranger to column-stores, however, we argue that query operators in column-stores can be redesigned in order to facilitate ad-hoc and planned sharing as a means to achieve better scalability.

Bibliography

Daniel J. Abadi. *Query execution in column-oriented database systems*. PhD thesis, Massachusetts Institute of Technology, 2008.

Daniel J. Abadi, Adam Marcus, Sammuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd Intenational Conference on Very Large Data Bases - VLDB 2007*, pages 411–422, 2007.

Michael Abraham. NAND Flash Trends for SSD/Enterprise. In *Flash Memory Summit*, Santa Clara, CA, USA, August 2010.

Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *Proceedings of the VLDB Endowment*, 2(1):361–372, 2009.

Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 147–160, Vancouver, Canada, June 2008.

Anastasia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving Relations for Cache Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases - VLDB 2001*, pages 169–180, September 2001.

Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Onyx: A Protoype Phase Change Memory Storage Array. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems - HotStorage 2011*, page 2, Portland, OR, USA, June 2011.

Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. Scaling up analytical queries with column-stores. In *Proceedings of the Sixth International Workshop on Testing Database Systems - DBTest 2013*, pages 8:1–8:6, New York, NY, USA, June 2013.

Nicole Alexander, Xavier Lopez, Siva Ravada, Susie Stephens, and Jack Wang. RDF Data Model in Oracle. *Oracle White Paper*, 2005.

Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable Bloom Filters. *Information Processing Letters*, 101(6):255–261, March 2007.

Christian Antognini. Bloom Filters in Oracle DBMS. *Oracle White Paper*, 2008.

Rolf Apweiler, Amos Bairoch, Cathy H. Wu, Winona C. Barker, Brigitte Boeckmann, Serenella Ferro, Elisabeth Gasteiger, Hongzhan Huang, Rodrigo Lopez, Michele Magrane, Maria J. Martin, Darren A. Natale, Claire O'Donovan, Nicole Redaschi, and Lai-Su L. Yeh. UniProt: the Universal Protein knowledgebase. *Nucleic acids research*, 32(Database Issue):D115–9, January 2004.

Subramanian Arumugam, Alin Dobra, Christopher M. Jermaine, Niketan Pansare, and Luis Perez. The DataPath system: a data-centric analytic processing engine for large data warehouses. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 519–530, Indianapolis, IN, USA, June 2010.

Manos Athanassoulis and Anastasia Ailamaki. BF-Tree: Approximate Tree Indexing. *Submitted for publication*, 2013.

Manos Athanassoulis, Anastasia Ailamaki, Shimin Chen, Phillip B. Gibbons, and Radu Stoica. Flash in a DBMS : Where and How ? *IEEE Data Engineering Bulletin*, 33(4):28–34, December 2010.

Manos Athanassoulis, Shimin Chen, Anastasia Ailamaki, Phillip B. Gibbons, and Radu Stoica. MaSM : Efficient Online Updates in Data Warehouses. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, Athens, Greece, June 2011.

Manos Athanassoulis, Bishwaranjan Bhattacharjee, Mustafa Canim, and Kenneth A. Ross. Path Processing using Solid State Storage. In *Proceedings of the 3rd International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2012*, Istanbul, Turkey, August 2012.

Manos Athanassoulis, Shimin Chen, Anastasia Ailamaki, Phillip B. Gibbons, and Radu Stoica. Online Updates on Data Warehouses via Judicious Use of Solid-State Storage. *Submitted for publication*, 2013.

Luiz Andre Barroso. Warehouse-scale Computing. In *Keynote talk at the 2010 ACM SIGMOD International Conference on Management of Data*, Indianapolis, IN, USA, June 2010.

Jacek Becla and Kian-Tat Lim. Report from the first workshop on extremely large databases (XLDB 2007). *Data Science Journal*, 7, February 2008.

Michael A. Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C. Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P. Spillane, and Erez Zadok. Don't Thrash: How to Cache Your Hash on Flash. *Proceedings of the VLDB Endowment*, 5(11): 1627–1637, July 2012.

Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, 24(2):1–10, May 1995.

Bishwaranjan Bhattacharjee, Kavitha Srinivas, and Octavian Udrea. Method and system to store RDF data in a relational store. Patent Application, 2011.

Christian Bizer and Andreas Schultz. The Berlin SPARQL Benchmark. *International Journal on Semantic Web and Information Systems*, 5(2), 2009.

Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems*, 5(3), 2009a.

Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - A crystallization point for the Web of Data. *Journal on Web Semantics*, 7(3):154–165, September 2009b.

Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

Peter Boncz. *Monet: A next-Generation DBMS Kernel For Query-Intensive Application*. PhD thesis, Universiteit van Amsterdam, 2002.

Dhruba Borthakur. Petabyte scale databases and storage systems at Facebook. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, page 1267, New York, NY, USA, June 2013.

Luc Bouganim, Björn Þór Jónsson, and Philippe Bonnet. uFLIP: Understanding Flash IO Patterns. In *Online Proceedings of the Fourth Biennial Conference on Innovative Data Systems Research - CIDR 2009*, Asilomar, CA, USA, January 2009.

Andrei Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1:636–646, 2002.

Geoffrey W. Burr, Bülent N. Kurdi, J. Campbell Scott, Chung Hon Lam, Kailash Gopalakrishnan, and Rohit S. Shenoy. Overview of Candidate Device Technologies for Storage-Class Memory. *IBM Journal of Research and Development*, 52(4):449–464, July 2008.

George Candea, Neoklis Polyzotis, and Radek Vingralek. A scalable, predictable join operator for highly concurrent data warehouses. *Proceedings of the VLDB Endowment*, 2(1):277–288, August 2009.

George Candea, Neoklis Polyzotis, and Radek Vingralek. Predictable performance and high query concurrency for data analytics. *The VLDB Journal*, 20(2):227–248, February 2011.

Mustafa Canim, Bishwaranjan Bhattacharjee, George A. Mihaila, Christian A. Lang, and Kenneth A. Ross. An Object Placement Advisor for DB2 Using Solid State Storage. *Proceedings of the VLDB Endowment*, 2(2):1318–1329, August 2009.

Mustafa Canim, Christian A. Lang, George A. Mihaila, and Kenneth A. Ross. Buffered Bloom filters on solid state storage. In *Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2010*, Singapore, Singapore, September 2010a.

Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. SSD Bufferpool Extensions for Database Systems. *Proceedings of the VLDB Endowment*, 3(1-2):1435–1446, September 2010b.

Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture - MICRO 2010*, pages 385–395, Atlanta, GA, USA, December 2010.

Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. *ACM SIGARCH Computer Architecture News*, 40(1):387–400, April 2012.

Sirish Chandrasekaran, Mehul A. Shah, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sammuel R. Madden, and Fred Reiss. TelegraphCQ: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, page 668, San Diego, CA, USA, June 2003.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - OSDI 2006*, Seattle, WA, USA, November 2006.

Fa-Chung Fred Chen and Margaret H. Dunham. Common subexpression processing in multiple-query processing. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):493–499, May 1998.

Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: a scalable continuous query system for Internet databases. *ACM SIGMOD Record*, 29(2):379–390, June 2000.

Shimin Chen. FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 73–86, Providence, RI, USA, 2009.

Shimin Chen, Phillip B. Gibbons, Todd C. Mowry, and Gary Valentin. Fractal prefetching B^+-Trees. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 157–168, Madison, WI, USA, June 2002.

Shimin Chen, Phillip B. Gibbons, and Suman Nath. PR-join: a non-blocking join achieving higher early result rate with statistical guarantees. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 147–158, Indianapolis, IN, USA, June 2010.

Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking Database Algorithms for Phase Change Memory. In *Online Proceedings of the Fifth Biennial Conference on Innovative Data Systems Research - CIDR 2011*, Asilomar, CA, USA, January 2011.

Hong-Tai Chou and David J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases - VLDB 1985*, pages 127–141, August 1985.

John Cieslewicz and Kenneth A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *Proceedings of the 33rd International Conference on Very Large Data Bases - VLDB 2007*, pages 339–350, Vienna, Austria, September 2007.

Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

Latha S. Colby, Richard L. Cole, Edward Haslam, Nasi Jazayeri, Galt Johnson, William J. McKenna, Lee Schumacher, and David Wilhite. Redbrick Vista: Aggregate Computation and Management. In *Proceedings of the Fourteenth International Conference on Data Engineering - ICDE 1998*, pages 174–177, Orlando, FL, USA, February 1998.

Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. Online Aggregation and Continuous Query Support in MapReduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 1115–1118, Indianapolis, IN, USA, June 2010.

Cathan Cook. Database Architecture: The Storage Engine. *MSDN Library*, 2001.

Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in multi-query optimization. *Journal of Computer and System Sciences*, 66(4):728–762, June 2003.

Data.gov. Online reference. *http://www.data.gov/*, 2013.

Data.gov.uk. Online reference. *http://www.data.gov.uk*, 2013.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation - OSDI 2004*, pages 137–150, San Francisco, CA, USA, December 2004.

Edward Doller. PCM and its impacts on memory hierarchy. *PDL Seminar, Carnegie Mellon University*, 2009.

Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and oranges: a comparison of RDF benchmarks and real RDF datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 145–156, Athens, Greece, June 2011.

Phillip M. Fernandez. Red Brick Warehouse: A Read-Mostly RDBMS for Open SMP Platforms. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, page 492, Minneapolis, MN, USA, May 1994.

David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A. Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John Prager, Nico Schlaefer, and Chris Welty. Building Watson: An Overview of the DeepQA Project. *AI Magazine*, 31(3):59–79, 2010.

Phil Francisco. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics. *IBM Redbooks*, 2011.

Richard Freitas. Storage Class Memory: Technology, Systems and Applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 985–986, Providence, RI, USA, July 2009.

FusionIO. ioDrive. *http://www.fusionio.com/products/iodrive/*, 2010.

FusionIO. ioDrive2. *http://www.fusionio.com/products/iodrive2/*, 2013.

Shen Gao, Jianliang Xu, Bingsheng He, Byron Choi, and Haibo Hu. PCMLogging: reducing transaction logging overhead with PCM. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management - CIKM 2011*, pages 2401–2404, Glasgow, Scotland, UK, October 2011.

Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. SharedDB: killing one thousand queries with one stone. *Proceedings of the VLDB Endowment*, 5(6):526–537, February 2012.

Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, February 1994.

Goetz Graefe. Sorting And Indexing With Partitioned B-Trees. In *Online Proceedings of the First Biennial Conference on Innovative Data Systems Research - CIDR 2003*, Asilomar, CA, USA, January 2003.

Goetz Graefe. B-tree indexes, interpolation search, and skew. In *Proceedings of the 2nd International Workshop on Data Management on New Hardware - DAMON 2006*, Chicago, IL, USA, June 2006a.

Goetz Graefe. B-tree indexes for high update rates. *ACM SIGMOD Record*, 35(1):39–44, March 2006b.

Goetz Graefe. Modern B-Tree Techniques. *Foundations and Trends in Databases*, 3(4):203–402, 2010.

Goetz Graefe and Harumi Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Proceedings of the 13th International Conference on Extending Database Technology - EDBT 2010*, pages 371–381, Lausanne, Switzerland, March 2010.

Goetz Graefe, Felix Halim, Stratos Idreos, Harumi Kuno, and Stefan Manegold. Concurrency control for adaptive indexing. *Proceedings of the VLDB Endowment*, 5(7):656–667, 2012.

Jim Gray. Tape is dead, Disk is Tape, Flash is Disk. *Gong show talk at the Third Biennial Conference on Innovative Data Systems Research - CIDR 2007*, 2007.

Andrey Gubichev and Thomas Neumann. Path Query Processing on Very Large RDF Graphs. In *Proceedings of the 14th International Workshop on the Web and Databases - WebDB 2011*, Athens, Greece, June 2011.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Journal on Web Semantics*, 3(2-3):158–182, October 2005.

Stavros Harizopoulos and Anastasia Ailamaki. A Case for Staged Database Systems. In *Online Proceedings of the First Biennial Conference on Innovative Data Systems Research - CIDR 2003*, Asilomar, CA, USA, January 2003.

Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastasia Ailamaki. QPipe: a simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 383–394, Baltimore, MD, USA, June 2005.

Oktie Hassanzadeh, Anastasios Kementsietsidis, and Yannis Velegrakis. Publishing Relational Data in the Semantic Web. In *Tutorial at the 8th Extended Semantic Web Conference - ESWC 2011*, Heraklion, Greece, May 2011.

HBase. Online reference. *http://hbase.apache.org/*, 2013.

Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. *Architecture of a Database System*. Now Publishers Inc., September 2007.

Sándor Héman, Marcin Zukowski, and Niels J. Nes. Positional update handling in column stores. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 543–554, 2010.

Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, Edwin Lewis-Kelham, Gerard de Melo, and Gerhard Weikum. YAGO2: exploring and querying world knowledge in time, space, context, and many languages. In *Proceedings of the 20th International Conference Companion on World Wide Web - WWW 2011*, pages 229–232, Hyderabad, India, March 2011.

Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: a spatially and temporally enhanced knowledge base from Wikipedia. *Artificial Intelligence*, 194:28–61, January 2013.

James Hughes. Revolutions in Storage. In *IEEE Conference on Massive Data Storage*, Long Beach, CA, May 2013.

IBM. Managing big data for smart grids and smart meters. *IBM White Paper*, 2012.

IBM. Keeping the Data in Your Warehouse Fresh. *IBM Data Magazine*, April 2013.

Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database Cracking. In *Online Proceedings of the Third Biennial Conference on Innovative Data Systems Research - CIDR 2007*, Asilomar, CA, USA, January 2007.

Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *Proceedings of the VLDB Endowment*, 4(9):586–597, June 2011.

William H. Inmon, R. H. Terdeman, Joyce Norris-Montanari, and Dan Meers. *Data Warehousing for E-Business*. John Wiley and Sons, 2003.

Intel. Intel X25-E SSD. *http://download.intel.com/design/flash/nand/extreme/319984.pdf*, 2009.

H. V. Jagadish, P. P. S. Narayan, Sridhar Seshadri, S. Sudarshan, and Rama Kanneganti. Incremental Organization for Data Recording and Warehousing. In *Proceedings of 23rd International Conference on Very Large Data Bases - VLDB 1997*, pages 16–25, Athens, Greece, August 1997.

Ryan Johnson, Stavros Harizopoulos, Nikos Hardavellas, Kivanc Sabirli, Ippokratis Pandis, Anastasia Ailamaki, Naju G. Mancheril, and Babak Falsafi. To Share or Not to Share? In *Proceedings of the 33rd International Conference on Very Large Data Bases - VLDB 2007*, pages 351–362, Vienna, Austria, September 2007.

Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database - EDBT 2009*, pages 24–35, Saint Petersburg, Russia, March 2009.

Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of 20th International Conference on Very Large Data Bases - VLDB 1994*, pages 439–450, September 1994.

Ju-Young Jung, Kelli Ireland, Jiannan Ouyang, Bruce Childers, Sangyeun Cho, Rami Melhem, Daniel Mosse, Jun Yang, Youtao Zhang, and A. J. Camber. Characterizing a Real PCM Storage Device (poster). In *2nd Annual Non-Volatile Memories Workshop 2011*, 2011.

Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, and Jin-Soo Kim. mu-tree: an ordered index structure for NAND flash memory. In *Proceedings of the 7th ACM & IEEE International conference on Embedded software - EMSOFT 2007*, pages 144–153, Salzburg, Austria, October 2007.

Martin L. Kersten, Stratos Idreos, Stefan Manegold, and Erietta Liarou. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. *Proceedings of the VLDB Endowment*, 4(12):1474–1477, 2011.

Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., 2nd edition, 2002.

Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., 1998.

Ioannis Koltsidas and Stratis D. Viglas. Flashing up the storage layer. *Proceedings of the VLDB Endowment*, 1(1):514–525, August 2008.

Sailesh Krishnamurthy, Michael J. Franklin, Joseph M. Hellerstein, and Garrett Jacobson. The case for precision sharing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - VLDB 2004*, pages 972–984, Toronto, Canada, August 2004.

Avinash Lakshman and Prashant Malik. Cassandra - A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35, April 2010.

Andrew Lamb, Matt Fuller, and Ramakrishna Varadarajan. The Vertica Analytic Database: C-Store 7 Years Later. *Proceedings of the VLDB Endowment*, 5(12):1790–1801, 2012.

Christian A. Lang, Bishwaranjan Bhattacharjee, Tim Malkemus, Sriram Padmanabhan, and Kwai Wong. Increasing Buffer-Locality for Multiple Relational Table Scans through Grouping and Throttling. In *Proceedings of the 23rd IEEE International Conference on Data Engineering - ICDE 2007*, pages 1136–1145, Istanbul, Turkey, 2007. Ieee.

Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1075–1086, Vancouver, Canada, June 2008.

Sang-Won Lee, Bongki Moon, Chanik Park, Joo Young Hwang, and Kangnyeon Kim. Accelerating In-Page Logging with Non-Volatile Memory. *IEEE Data Engineering Bulletin*, 33(4): 41–47, December 2010.

Yinan Li, Bingsheng He, Jun Yang, Qiong Luo, Ke Yi, and Robin Jun Yang. Tree Indexing on Solid State Drives. *Proceedings of the VLDB Endowment*, 3(1-2):1195–1206, September 2010.

Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles - SOSP 2011*, pages 1–13, Cascais, Portugal, October 2011.

Hyesook Lim and So Yeon Kim. Tuple Pruning using Bloom Filters for packet classification. *IEEE Micro*, 30(3):48–59, May 2010.

Witold Litwin and David B. Lomet. The Bounded Disorder Access Method. In *Proceedings of the Second International Conference on Data Engineering - ICDE 1986*, pages 38–48, Los Angeles, CA, USA, February 1986.

Zhen Liu, Srinivasan Parthasarathy, Anand Ranganathan, and Hao Yang. Near-optimal algorithms for shared filter evaluation in data stream systems. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 133–146, Vancouver, Canada, June 2008.

Guanlin Lu, Biplob Debnath, and David H. C. Du. A Forest-structured Bloom Filter with flash memory. In *Proceedings of the 27th IEEE Symposium on Mass Storage Systems and Technologies - MSST 2011*, pages 1–6, Denver, CO, USA, May 2011.

Sammuel R. Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, page 49, Madison, WI, USA, June 2002.

Anirban Majumder, Rajeev Rastogi, and Sriram Vanama. Scalable regular expression matching on data streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 161–172, Vancouver, Canada, June 2008.

Mitzi McCarthy and Zhen He. Efficient Updates for OLAP Range Queries on Flash Memory. *The Computer Journal*, 54(11):1773–1789, February 2011.

Nimrod Megiddo and Dharmendra S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies - FAST 2003*, pages 115–130, San Francisco, CA, USA, April 2003.

Manish Mehta, Valery Soloviev, and David J. DeWitt. Batch Scheduling in Parallel Database Systems. In *Proceedings of the Ninth International Conference on Data Engineering - ICDE 1993*, pages 400–410, Vienna, Austria, April 1993.

Guido Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of 24th International Conference on Very Large Data Bases - VLDB 1998*, pages 476–487, New York, NY, USA, August 1998.

MonetDB. Online reference. *http://www.monetdb.org*, 2013.

Mark Morri. Teradata Multi-Value Compression V2R5.0. *Teradata White Paper*, July 2002.

James K Mullin. Optimal Semijoins for Distributed Database Systems. *IEEE Transactions on Software Engineering*, 16(5):558–560, May 1990.

Gap-Joo Na, Bongki Moon, and Sang-Won Lee. IPLB$^+$-tree for Flash Memory Database Systemsa. *Journal of Information Science and Engineering*, 27(1):111–127, January 2011.

Thomas Neumann and Gerhard Weikum. RDF-3X: a RISC-style engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, August 2008.

Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 297–306, Washington, DC, USA, May 1993.

Patrick E. O'Neil. Model 204 Architecture and Performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems - HPTS 1987*, pages 40–59, Asilomar, CA, USA, September 1987.

Patrick E. O'Neil. The SB-Tree: An Index-Sequential Structure for High-Performance Sequential Access. *Acta Informatica*, 29(3):241–265, June 1992.

Patrick E. O'Neil and Dallan Quass. Improved query performance with variant indexes. *ACM SIGMOD Record*, 26(2):38–49, June 1997.

Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

Patrick E. O'Neil, Elizabeth J. O'Neil, and Xuedong Chen. Star Schema Benchmark. Technical report, UMass/Boston, 2009.

Oracle. On-time data warehousing with oracle10g - information at the speed of your business. *Oracle White Paper*, 2004.

Oracle. Oracle Database 12c for Data Warehousing and Big Data. *Oracle White Paper*, 2013.

Ovonyx. Technology Presentation - Ovonic Unified Memory, page 29. *http://ovonyx.com/technology/technology.pdf*, 2007.

Oguzhan Ozmen, Kenneth Salem, Jiri Schindler, and Steve Daniel. Workload-aware Storage Layout for Database Systems. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 939–950, Indianapolis, IN, USA, June 2010.

Nikolaos Papandreou, Haralampos Pozidis, Aggeliki Pantazi, Abu Sebastian, Matthew J. Breitwisch, Chung Hon Lam, and Evangelos Eleftheriou. Programming algorithms for multilevel phase-change memory. In *Proceedings of the International Symposium on Circuits and Systems - ISCAS 2011*, pages 329–332, Rio de Janeiro, Brazil, May 2011.

Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search—a log logN search. *Communications of the ACM*, 21(7):550–553, July 1978.

Neoklis Polyzotis, Spiros Skiadopoulos, Panos Vassiliadis, Alkis Simitsis, and Nils-Erik Frantzell. Meshing Streaming Updates with Persistent Data in an Active Data Warehouse. *IEEE Transactions on Knowledge and Data Engineering*, 20(7):976–991, July 2008.

Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. Sharing data and work across concurrent analytical queries. *Proceedings of the VLDB Endowment*, 6(9):637–648, July 2013.

Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. Main-memory scan sharing for multi-core CPUs. *Proceedings of the VLDB Endowment*, 1(1): 610–621, August 2008.

Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 3rd edition, 2002.

RDf. Resource Description Framework. *http://www.w3.org/RDF/*, 2013.

Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. B^+-Tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. *Proceedings of the VLDB Endowment*, 5(4):286–297, December 2011.

Christian Esteve Rothenberg, Carlos Macapuna, Fabio Verdi, and Mauricio Magalhaes. The deletable Bloom filter: a new member of the Bloom family. *IEEE Communications Letters*, 14 (6):557–559, June 2010.

Nicholas Roussopoulos. View indexing in relational databases. *ACM Transactions on Database Systems*, 7(2):258–290, June 1982.

Prasan Roy, Sridhar Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. *ACM SIGMOD Record*, 29(2):249–260, June 2000.

Philip Russom. High-Performance Data Warehousing. *TDWI Best Practices Report*, 2012.

Jiri Schindler, Sandip Shete, and Keith A. Smith. Improving throughput for small disk requests with proximal I/O. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies - FAST 2011*, pages 10–10, San Jose, CA, USA, February 2011.

Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP^2Bench: A SPARQL Performance Benchmark. In *Proceedings of the 25th International Conference on Data Engineering - ICDE 2009*, pages 222–233, Shanghai, China, April 2009.

Timos K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1): 23–52, March 1988.

Dennis G. Severance and Guy M. Lohman. Differential files: their application to the maintenance of large databases. *ACM Transactions on Database Systems*, 1(3):256–267, September 1976.

Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The Semantic Web Revisited. *IEEE Intelligent Systems*, 21(3):96–101, May 2006.

Junho Shim, Peter Scheuermann, and Radek Vingralek. Dynamic caching of query results for decision support systems. In *Proceedings of the 11th International Conference on Scientific and Statistical Database Management - SSDBM 1999*, pages 254–263, Cleveland, OH, USA, July 1999.

Lefteris Sidirourgos and Martin L. Kersten. Column Imprints: A Secondary Index Structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 893–904, New York, NY, USA, June 2013.

Alkis Simitsis, Kevin Wilkinson, Malu Castellanos, and Umeshwar Dayal. QoX-driven ETL design: reducing the cost of ETL consulting engagements. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 953–960, Providence, RI, USA, July 2009.

Radu Stoica, Manos Athanassoulis, Ryan Johnson, and Anastasia Ailamaki. Evaluating and repairing write performance on flash devices. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware - DAMON 2009*, pages 9–14, Providence, RI, USA, June 2009.

Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sammuel R. Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented DBMS. In *Proceedings of the 31st International Conference on Very Large Data Bases - VLDB 2005*, pages 553–564, Trondheim, Norway, August 2005.

Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The missing memristor found. *Nature*, 453(7191):80–83, May 2008.

Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and Practice of Bloom Filters for Distributed Systems. *IEEE Communications Serveys & Tutorials*, 14(1): 131–155, 2012.

Jason Taylor. Flash at Facebook: The Current State and Future Potential. In *Keynote talk at the Flash Memory Summit*, Santa Clara, CA, USA, August 2013.

TPC. Specification of TPC-H benchmark. *http://www.tpc.org/tpch/*, 2013.

Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 59–72, Providence, RI, USA, July 2009.

Philipp Unterbrunner, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. Predictable performance for unpredictable workloads. *Proceedings of the VLDB Endowment*, 2(1):706–717, August 2009.

Vertica. Online reference. *http://www.vertica.com*, 2013.

Virtuoso. Online reference. *http://www.openlinksw.com/dataspace/dav/wiki/Main/VOSRDF*, 2011.

Colin White. Intelligent business strategies: Real-time data warehousing heats up. *DM Review*, August 2002.

Kevin Wilkinson. Jena property table implementation. In *Proceedings of the Second International Workshop on Scalable Semantic Web Knowledge Base - SSWS 2006*, Athens, GA, Greece, November 2006.

Jingren Zhou, Per-Ake Larson, and Hicham G. Elmongui. Lazy Maintenance of Materialized Views. In *Proceedings of the 33rd International Conference on Very Large Data Bases - VLDB 2007*, pages 231–242, Vienna, Austria, September 2007.

Paul Zikopoulos, Dirk DeRoos, Krishnan Parasuraman, Thomas Deutsch, David Corrigan, and James Giles. *Harness the Power of Big Data – The IBM Big Data Platform.* Mcgraw-Hill, 2012.

Marcin Zukowski, Sándor Héman, Niels J. Nes, and Peter Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *Proceedings of the 33rd International Conference on Very Large Data Bases - VLDB 2007*, pages 723–734, Vienna, Austria, September 2007.

Manoussos-Gavriil (Manos) Athanassoulis

Ph.D. in Computer Science
Data-Intensive Applications and Systems laboratory
School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne
CH – 1015, Lausanne, Switzerland
manos.athanassoulis@epfl.ch
http://manos.athanassoulis.net

RESEARCH INTERESTS

big data, data analytics, data management systems, database systems design using modern storage technologies and multicore processors

ACADEMIC BACKGROUND

Ph.D. in Computer Science, December 2013 **Ecole Polytechnique Fédérale de Lausanne**

Thesis: "Solid-State Storage and Work Sharing for Efficient Scaleup Data Analytics"
Advisor: Prof. Anastasia Ailamaki (natassa@epfl.ch)

M.Sc. in Computer Systems Technology, March 2008 **University of Athens, Greece**

Thesis: "Thread scheduling technique for avoiding page thrashing"
Supervisor: Prof. Stathes Hadjiefthymiades (shadj@di.uoa.gr)

B.Sc. in Informatics & Telecommunications, September 2005 **University of Athens, Greece**

Thesis: "A Multi-Criteria Message Forwarding Architecture for Wireless Sensor Networks"
Supervisor: Prof. Stathes Hadjiefthymiades (shadj@di.uoa.gr)

HONORS & AWARDS

- EPFL Outstanding Teaching Assistantship Award 2013
- IBM Ph.D. Fellowship, 2011-2012
- Prize for having the best record, M.Sc., University of Athens, 2007
- Graduate Fellowship from "Public Benefit Foundation Alexander S. Onassis", 2006-2007
- Valedictorian B.Sc., University of Athens, 2005
- Scholarship & Prize for the best record each year, B.Sc., University of Athens, 2002-2005

PUBLICATIONS

Data Management Systems:

- I. Psaroudakis, M. Athanassoulis, A. Ailamaki, "Sharing Data and Work Across Concurrent Analytical Queries", In Proc. of Very Large Databases (**VLDB**), Trento, Italy, August 26-30, 2013.
- I. Alagiannis, M. Athanassoulis, and A. Ailamaki, "Scaling Up Analytical Queries with Column-Stores", 6th International Workshop on Testing Database Systems, (**DBTest**), New York, NY, June 24, 2013.
- M. Athanassoulis, B. Bhattacharjee, M. Canim, K. A. Ross, "Querying Persistent Graphs using Solid State Storage", 4th Annual Non-Volatile Memories Workshop (**NVMW**), San Diego, CA, March 3-5, 2013.
- M. Athanassoulis, B. Bhattacharjee, M. Canim, K. A. Ross, "Path Processing Using Solid State Storage", In Proc. of Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (**ADMS**), Istanbul, Turkey, August 27, 2012.
- R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. "Scalability of write-ahead logging on

multicore and multisocket hardware", The VLDB Journal (**VLDBJ**), Vol. 21 (2), 2012.

- M. Athanassoulis, S. Chen, A. Ailamaki, P. Gibbons, and R. Stoica, "MaSM: Efficient Online Updates in Data Warehouses", In Proc. of Proceedings of the ACM SIGMOD International Conference on Management of Data (**SIGMOD**), Athens, Greece, June 12-16, 2011.
- M. Athanassoulis, A. Ailamaki, S. Chen, P. Gibbons, and R. Stoica, "Flash in a DBMS: Where and How?", **IEEE Data Engineering Bulletin** 2010, Vol. 33(4).
- S. Chen, A. Ailamaki, M. Athanassoulis, P. Gibbons, R. Johnson, I. Pandis and R. Stoica, "TPC-E vs. TPC-C: Characterizing the New TPC-E Benchmark via an I/O Comparison Study", **SIGMOD Record**, 2010, Vol. 39(4).
- R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki, "Aether: A Scalable Approach to Logging", In Proc. of Very Large Databases (**VLDB**), Singapore, September 13-17, 2010.

Invited for publication at the Special Issue of VLDB Journal for **VLDB 2010 Best Papers**.

- R. Johnson, M. Athanassoulis, R. Stoica and A. Ailamaki, "A New Look at the Roles of Spinning and Blocking", In Proc. of Workshop on Data Management on New Hardware (**DaMoN**), Providence, RI, USA, June 28, 2009.
- R. Stoica, M. Athanassoulis, R. Johnson and A. Ailamaki, "Evaluating and Repairing Write Performance on Flash Devices". In Proc. of Workshop on Data Management on New Hardware (**DaMoN**), Providence, RI, USA, June 28, 2009.

Wireless Sensor Networks:

- M. Athanassoulis, I. Alagiannis and S. Hadjiefthymiades, "Energy Efficiency in Wireless Sensor Networks: A Utility-Based Architecture", In Proc. of the 13th European Wireless (**EW**), Paris, France, April 1-4, 2007.
- M. Athanassoulis, I. Alagiannis and S. Hadjiefthymiades, "A Multi-Criteria Message Forwarding Architecture for Wireless Sensor Networks", In Proc. of the 10th Pan-Hellenic Conference on Informatics (**PCI**), Volos, Greece, November 11-13, 2005.

WORKING EXPERIENCE

10/2008 – Present **Ecole Polytechnique Fédérale de Lausanne** **Lausanne, Switzerland**

Research Assistant & Member of the Data-Intensive Applications and Systems (DIAS) laboratory, advised by Professor Anastasia Ailamaki, on exploiting modern storage technologies (solid-state storage) to enable features and improve performance of data management and data analysis systems.

7/2011 – 10/2011 **IBM Watson Research Labs** **Hawthorne, NY, USA**

Summer Intern at the Database Research Group, mentored by Bishwaranjan Bhattacharjee and prof. Kenneth Ross.

10/2006 – 8/2008 **University of Athens** **Athens, Greece**

Junior Research Assistant in EU-funded project: SCIER - A technological simulation platform to managing natural hazards (http://www.scier.eu).

1/2004 – 9/2006 **National Technical University of Athens** **Athens, Greece**

Junior Research Assistant in several EU-funded projects, responsible for maintaining and updating documentation and manuals.

TEACHING

Spring 2012, 2013	EPFL. TA in undergraduate course *Introduction to Database Systems*. Instructor: Prof. Anastasia Ailamaki (natassa@epfl.ch).
Spring 2010, 2011	EPFL. TA in undergraduate course *Databases*. Instructor: Prof. Anastasia Ailamaki (natassa@epfl.ch).
Spring 2006, 2007	University of Athens. TA in graduate course *Foundations of Databases*. Instructor: Prof. Manolis Koumparakis (koubarak@di.uoa.gr).
Fall 2006	University of Athens. TA in undergraduate course *Computer Architecture I*. Instructor: Prof. Antonis Paschalis (paschali@di.uoa.gr).
Fall 2005, 2006	University of Athens. TA in undergraduate course *Operating Systems*. Instructor: Prof. Stathes Hadjiefthymiades (shadj@di.uoa.gr).

REVIEWING

Reviewer	ACM Transactions on Storage The Computer Journal Information Systems
External Reviewer	SIGMOD 2014 EDBT 2014 ICDE 2014 VLDB 2011, 2013 ASPLOS 2012 DAMON 2009, 2010

LANGUAGES

Greek	Native
English	Fluent – Cambridge Certificate of Proficiency in English (CPE), 1999
French	Intermediate – DELF I, 2003

Last update: Jan 2014.