TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik
Lehrstuhl für Datenbanksysteme

Advancing Analytical Database Systems

Andreas Michael Kipf

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr. Florian Matthes
Prüfer der Dissertation:	1.	Prof. Alfons Kemper, Ph.D.
	2.	Prof. Divyakant Agrawal, Ph.D. (University of California at Santa Barbara)
	3.	Prof. Dr. Thomas Neumann

Die Dissertation wurde am 11.12.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 13.01.2020 angenommen.

Abstract

Over the last decade, we have seen a rise in demand for analytical data processing. This trend is driven by an increase in volume and velocity of data that is being created and by companies seeking to unlock its potential. Examples include vehicle telemetry, health, and industrial data. At the same time, advances in hardware and machine learning enable system builders to create ever faster and smarter database systems.

This thesis makes three contributions to the design and implementation of such systems. First, we compare main-memory databases with modern streaming systems using a telecommunications benchmark, identify performance and usability gaps, and explore extensions to database systems. These extensions include user-space networking for faster client-server communication and a scale-out architecture. Second, we propose an approach to processing geospatial point data in memory. In particular, we contribute a novel polygon index that allows for efficient point-polygon joins. This index can either provide precision-bounded approximate results or can be used to identify true and candidate join pairs. Most notably, it can learn from historical data to become more effective. Third, we contribute a new deep learning approach to cardinality estimation, which is the core problem in cost-based query optimization. We propose a new neural network model that can capture correlations between columns, even across tables. Trained with past queries, our model can predict the cardinalities of future queries and significantly enhances the quality of cardinality estimation.

Zusammenfassung

Während des letzten Jahrzehnts hat sich die Nachfrage nach analytischer Datenverarbeitung stets gesteigert. Daten werden in immer größeren Mengen und mit zunehmender Geschwindigkeit generiert und Unternehmen streben danach deren Potential auszuschöpfen. Als Beispiele sind Daten von Fahrzeugen, aus dem Gesundheitswesen sowie aus der Industrie zu nennen. Gleichzeitig ermöglichen es Fortschritte in den Bereichen Hardware und Maschinelles Lernen immer schnellere und intelligentere Datenbanksysteme zu entwickeln.

Diese Arbeit trägt zum aktuellen Forschungsstand von analytischen Datenbanksystemen bei. Wir identifizieren und explorieren Erweiterungen um Analysen auf Eventströmen besser zu unterstützen. Insbesondere vergleichen wir Hauptspeicherdatenbanksysteme mit modernen Streamingsystemen anhand eines Telekommunikationsbenchmarks. Als Erweiterungen untersuchen wir eine User-Space-Bibliothek zur effizienteren Netzwerkkommunikation und eine verteilte Datenbankarchitektur. Darüber hinaus stellen wir eine neue Indexstruktur für Polygone vor, die eine effiziente Verarbeitung von Geodaten im Hauptspeicher ermöglicht. Diese Datenstruktur kann je nach Anwendung approximative als auch exakte Ergebnisse liefern. Erwähnenswert ist, dass unser Index von historischen Daten lernen kann, um seine Effektivität zu steigern. Zudem präsentieren wir einen neuen Ansatz für Kardinalitätsschätzungen mittels maschinellen Lernens. Konkret schlagen wir ein neues neuronales Netzwerk vor, das Korrelationen zwischen Spalten abbilden kann und dies sogar über mehrere Tabellen hinweg. Wir trainieren dieses Netzwerk mit vergangenen Anfragen, wodurch es lernt die Kardinalitäten zukünftiger Anfragen mit hoher Genauigkeit vorherzusagen.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, Prof. Alfons Kemper. He gave me the freedom to pursue a variety of ideas and was always there when I needed his advice. I would also like to thank Prof. Thomas Neumann for his continuous feedback and inspiration.

A special thank you goes to Prof. Divyakant Agrawal from UC Santa Barbara for being part of my thesis committee. I also thank Prof. Florian Matthes for chairing my doctoral examination.

As a member of the TUM database group, I had the opportunity to work and make friends with many great people: Alexander van Renen, Varun Pandey (thank you for being an amazing office mate), Harald Lang, Linnea Passing (thank you for sharing the Software Campus experience with me), Jan Böttcher (thank you for teaching me how to swim freestyle), Timo Kersten, Tobias Mühlbauer, Wolf Rödiger, Manuel Then, Moritz Kaufmann, Jan Finis, Viktor Leis, Dimitri Vorona, Nina Hubig, Bernhard Radke, André Kohn, Michael Freitag, Christian Winter, Christoph Anneser, Dominik Durner, Maximilian Schüle, Lukas Vogel, Moritz Sichert, Philipp Fent, Maximilian Bandle, Michael Haubenschild, Adrian Vogelsgesang, and many more. Thank you, Angelika Reiser, for your guidance and feedback.

I am also very grateful to have had the opportunity to collaborate with fantastic researchers outside of TUM: Prof. Peter Boncz from CWI Amsterdam contributed to most of my projects and is a great source of inspiration to me. Eleni Tzirita Zacharatou from EPFL (now at TU Berlin) worked with me on geospatial data processing and Lucas Braun from ETH (now at Oracle) was of great help in our effort on analytics on fast data.

In the summers of 2017 and 2018, I interned at Google in Mountain View and in Zurich. Thank you to my hosts and co-hosts Jagan Sankaranarayanan, Kevin Lai, Damian Chromejko, and Alexander Hall for your inspiration and mentorship. It was a great experience that I would not want to miss.

Finally, I would like to thank my family. To my partner, Vanessa, for taking on this journey with me. Your support over the last twelve years has been tremendous. To my brother, Thomas, for the endless inspiration and for always being there (even as a co-author). To my mother, Marianne, for your outstanding support. I dedicate this thesis to my father, Harald, who passed away when I started at TUM. He taught me to pursue my goals with passion and to never give up.

Funding. This work has been partially supported by the German Federal Ministry of Education and Research (BMBF) grant `01IS12057` (FASTDATA). It is further part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

PREFACE

Excerpts of this thesis have been published in advance.

Chapter 2 is drawn from the following publications with modifications to the description and evaluation of "HyPerParallel":

Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. "Analytics on Fast Data: Main-Memory Database Systems versus Modern Streaming Systems". In: *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.* Pp. 49–60

An extended version appeared in ACM TODS (Best of EDBT 2017):
Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. "Scalable Analytics on Fast Data". In: *ACM Trans. Database Syst.* 44.1 (2019), 1:1–1:35

Chapter 3 is drawn from the following publications with minor modifications:

Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Approximate Geospatial Joins with Precision Guarantees". In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pp. 1360–1363

An extended version appeared in EDBT 2020:
Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Christoph Anneser, Eleni Tzirita Zacharatou, Harish Doraiswamy, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins". In: *Proceedings of the 23nd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pp. 347–358

Chapter 4 is a consolidation of the following publications:

Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. "Learned Cardinalities: Estimating Correlated Joins with Deep Learning". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*

Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Estimating Cardinalities with Deep Sketches". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* Pp. 1937–1940

Andreas Kipf, Michael Freitag, Dimitri Vorona, Peter Boncz, Thomas Neumann, and Alfons Kemper. "Estimating Filtered Group-By Queries is Hard: Deep Learning to the Rescue". In: *1st International Workshop on Applied AI for Database Systems and Applications* (2019)

Chapters 1 and 5 also draw from these publications, but also contain novel, unpublished material. In addition to these publications, the author of this thesis also co-authored the following related work, which is not part of this thesis:

Varun Pandey, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper. "High-Performance Geospatial Analytics in HyPerSpace". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pp. 2145–2148

Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. "How Good Are Modern Spatial Analytics Systems?" In: *PVLDB* 11.11 (2018), pp. 1661–1673

Harald Lang, Andreas Kipf, Linnea Passing, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines". In: *Proceedings of the 14th International Workshop on Data Management on New Hardware, Houston, TX, USA, June 11, 2018*, 5:1–5:8

Harald Lang, Linnea Passing, Andreas Kipf, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines". In: *VLDBJ* (2019), pp. 1–18

Christian Winter, Andreas Kipf, Thomas Neumann, and Alfons Kemper. "GeoBlocks: A Query-Driven Storage Layout for Geospatial Data". In: *CoRR* abs/1908.07753 (2019)

Dimitri Vorona, Andreas Kipf, Thomas Neumann, and Alfons Kemper. "DeepSPACE: Approximate Geospatial Query Processing with Deep Learning". In: *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2019, Chicago, IL, USA, November 5-8, 2019*, pp. 500–503

Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. "SOSD: A Benchmark for Learned Indexes". In: *NeurIPS Workshop on Machine Learning for Systems* (2019)

Christoph Anneser, Andreas Kipf, Harald Lang, Thomas Neumann, and Alfons Kemper. "The Case for Hybrid Succinct Data Structures". In: *Proceedings of the 23nd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pp. 391–394

Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory". In: *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*

Philipp Fent, Michael Jungmair, Andreas Kipf, and Thomas Neumann. "START — Self-Tuning Adaptive Radix Tree". In: *36th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2020, Dallas, TX, USA, April 20-24, 2020*, pp. 147–153

All of the publications listed above are marked with an asterisk (*) in the bibliography in compliance with § 6 Abs. 6 Satz 3 Promotionsordnung der Technischen Universität München.

CONTENTS

ACKNOWLEDGMENTS i
PREFACE iii
1 INTRODUCTION 1
1.1 Analytical Database Systems . . . 1
1.1.1 Challenges . . . 1
1.1.2 Research Opportunities . . . 4
1.2 The Learned Systems Era . . . 6
1.3 Contributions . . . 8
2 ANALYTICS ON FAST DATA 11
2.1 Introduction . . . 11
2.2 Approaches . . . 14
2.2.1 Main-Memory Database Systems . . . 14
2.2.2 Modern Streaming Systems . . . 15
2.2.3 AIM . . . 17
2.2.4 Summary . . . 17
2.3 Workload . . . 21
2.3.1 Description . . . 21
2.3.2 Implementations . . . 23
2.4 Extensions to MMDBs . . . 27
2.4.1 Mitigating the Network Bottleneck . . . 27
2.4.2 Improving OLTP Capabilities . . . 29
2.4.3 Scaling out Using Horizontal Partitioning . . . 30
2.5 Performance Evaluation . . . 32
2.5.1 Configuration . . . 32
2.5.2 Overall Performance . . . 33
2.5.3 Read Performance . . . 34
2.5.4 Write Performance . . . 34
2.5.5 Query Response Times . . . 36
2.5.6 Impact of Number of Clients . . . 36
2.5.7 Impact of Number of Aggregates . . . 37
2.5.8 Impact of Skew . . . 38
2.5.9 Impact of User-Space Networking . . . 40
2.5.10 Distributed Setting . . . 42
2.6 Closing the Gap . . . 44
2.7 Conclusions . . . 46
3 GEOSPATIAL ANALYTICS 47
3.1 Introduction . . . 47
3.2 Background . . . 49

3.2.1 Location Discretization 49
3.2.2 Polygon Approximations 50
3.2.3 PIP Test 51
3.3 Approach 51
3.3.1 Adaptive Cell Trie (ACT) Indexing 52
3.3.2 Approximate Join with Precision Bound 60
3.3.3 Accurate Join 61
3.3.4 Implementation Details 62
3.4 Experimental Evaluation 63
3.4.1 Infrastructure 64
3.4.2 Datasets and Queries 64
3.4.3 Polygon Approximations 65
3.4.4 Approximate Join 65
3.4.5 Accurate Join 72
3.4.6 Comparison with GPU Algorithms 76
3.5 Related Work 77
3.5.1 Spatial Join Techniques 77
3.5.2 Systems 78
3.5.3 Modern Hardware 78
3.6 Conclusions 78
4 LEARNED CARDINALITIES 79
4.1 Introduction 79
4.2 Approach 80
4.2.1 Set-Based Query Representation 81
4.2.2 Model 81
4.2.3 Generating Training Data 84
4.2.4 Enriching the Training Data 85
4.2.5 Training and Inference 85
4.3 Evaluation 85
4.3.1 Estimation Quality 87
4.3.2 0-Tuple Situations 88
4.3.3 Removing Model Features 88
4.3.4 Generalizing to More Joins 89
4.3.5 JOB-light 90
4.3.6 Hyperparameter Tuning 91
4.3.7 Model Costs 91
4.3.8 Optimization Metrics 92
4.4 Filtered Group-By Queries 93
4.4.1 Problem 93
4.4.2 Applications 94
4.4.3 Adapting Our Model 94
4.4.4 Results 96
4.5 Optimizer Integration 103

4.5.1 Overview 103
4.5.2 Training and Query Flow 104
4.5.3 Web Interface 106
4.5.4 Open Challenges 107
4.6 Discussion 107
4.6.1 Generalization 107
4.6.2 Adaptive Training 108
4.6.3 Strings 108
4.6.4 Complex Predicates 108
4.6.5 More Bitmaps 108
4.6.6 Filtered Group-By Estimates 109
4.6.7 Uncertainty Estimation 110
4.6.8 Updates 110
4.7 Related Work 111
4.7.1 ML-Based Approaches 111
4.7.2 Sampling 111
4.7.3 Group-By Estimates 112
4.8 Conclusions 113
5 FUTURE WORK 115
BIBLIOGRAPHY 117

LIST OF FIGURES

Figure 1 HyPerMaps. Interactive exploration of changing point datasets. . . . 2

Figure 2 Runtimes of TPC-H Query 5 (SF1) with random query plans. Plot adapted from [152]. Numbers reproduced on AMD Ryzen Threadripper 1950X. . . . 3

Figure 3 GFLOPS (32-bit) of various GPUs. Data collected by [182]. 7

Figure 4 Analytics on fast data. Streaming events may be processed concurrently in different partitions, whereas analytical queries cross partition boundaries and require a consistent state. . . . 12

Figure 5 The Huawei-AIM workload. . . . 23

Figure 6 Hybrid processing in Flink. A CoFlatMap operator interleaves events with analytical queries. . . . 26

Figure 7 Using mTCP and DPDK on the server side to accelerate message throughput. . . . 29

Figure 8 HyPerDistributed with multiple horizontal partitions (shards). Events are dispatched to the corresponding partition. Queries are sent to all partitions and aggregated on the client side. . . . 31

Figure 9 Analytical query throughput for 10 M subscribers at 10,000 events/s. . . . 33

Figure 10 Analytical query throughput for 10 M subscribers. . . . 34

Figure 11 Event processing throughput with an increasing number of event processing threads. . . . 35

Figure 12 Event processing throughput with an increasing number of event processing threads (using optimistic locking in HyPerParallel). . . . 35

Figure 13 Query response times. . . . 37

Figure 14 Analytical query throughput with an increasing number of clients. . . . 37

Figure 15 Analytical query throughput for 10 M subscribers and 42 aggregates at 10,000 events/s. . . . 38

Figure 16 Event processing throughput for 42 aggregates with an increasing number of event processing threads. . . . 39

Figure 17 Relative event processing performance for 546 aggregates with increasing skew using 10 event processing threads (Zipf factor = 0 represents a uniform distribution). . . . 40
Figure 18 Number of round trips in K messages/s for two different payload sizes with an increasing number of client threads (mTCP/mTCP = server and clients use mTCP, mTCP/-TCP = server uses mTCP and clients TCP, and TCP/TCP = server and clients use TCP). . . . 41
Figure 19 Number of round trips in K messages/s for two different client thread counts with an increasing number of updated columns. . . . 42
Figure 20 Analytical query throughput for 10 M subscribers with an increasing number of nodes. . . . 43
Figure 21 Overview of possible extensions to MMDBs and their benefits. . . . 44
Figure 22 Polygons are indexed in a new trie data structure that is probed with dynamic points. The join result can be aggregated or materialized. . . . 48
Figure 23 Quadtree-based cell decomposition and Hilbert curve-based enumeration. . . . 50
Figure 24 A covering (blue cells) and an interior covering (green cells) of a polygon. . . . 51
Figure 25 A combined covering may be less selective than two individual coverings. The arrows indicate that the cells will be expanded. . . . 53
Figure 26 Precision preserving conflict resolution. c_1 is marked in blue, c_2 in green, and the cells in d in purple. Note that c_1 contains c_2. . . . 54
Figure 27 A super covering of neighborhoods in NYC's Jamaica Bay. 56
Figure 28 Adaptive Cell Trie indexing three polygons a, b, and c. Here, ACT uses two bits per level. In practice, we use up to eight bits (a fanout of 255) to reduce the tree height. Note that the figure only shows the cell rasterization for the part of the map that corresponds to the radix tree. . . . 59
Figure 29 Single-threaded throughput of our approximate algorithm with different data structures (4 m precision). . . . 68
Figure 30 Single-threaded throughput of our approximate algorithm with different precisions and data structures (neighborhood polygons). . . . 70
Figure 31 Multi-threaded throughput of our approximate algorithm with different data structures (neighborhood polygons, 4 m precision). . . . 71

Figure 32 Single-threaded throughput of our approximate algorithm (4 m precision) with uniform point data. 72
Figure 33 Single-threaded throughput of our approximate algorithm (Twitter datasets, polygon counts in brackets). . . . 73
Figure 34 Single-threaded throughput of our accurate algorithm (with different ACT fanouts) compared to S2ShapeIndex (with 1 and 10 edges per cell) and the R-tree. 74
Figure 35 Throughput of ACT4 (16 threads) compared to the two GPU algorithms on AWS (GPU = Bounded Raster Join for 15 m and 4 m and Accurate Raster Join for exact). . . 76
Figure 36 Architecture of our multi-set convolutional network. Tables, joins, and predicates are represented as separate modules, comprised of one two-layer neural network per set element with shared parameters. Module outputs are averaged, concatenated, and fed into a final output network. 82
Figure 37 Query featurization as sets of feature vectors. 83
Figure 38 Estimation errors on the synthetic workload. The box boundaries are at the 25th/75th percentiles and the horizontal "whisker" lines mark the 95th percentiles. 87
Figure 39 Estimation errors on the synthetic workload with different model variants. 89
Figure 40 Estimation errors on the scale workload showing how MSCN generalizes to queries with more joins. 90
Figure 41 Convergence of the mean q-error on the validation set with the number of epochs. 92
Figure 42 Architecture of the adapted MSCN model. Tables, group-by columns, and predicates are represented as separate MLP modules that provide input to a final output network that predicts query cardinalities. 96
Figure 43 Estimation errors on the query workload. The box boundaries are at the 25th/75th percentiles and the horizontal "whisker" lines mark the 95th percentiles. 99
Figure 44 Estimation errors on the query workload with an increasing number of training queries. 100
Figure 45 Cardinality estimates (y-axis) of PostgreSQL, HyPer, SC-BC, and Deep Sketches. Selection on `production_year` (x-axis) and group-by on `kind_id` and/or `phonetic_code`. 101
Figure 46 Creation and usage of a Deep Sketch. Depending on the number of training queries, training can be expensive. However, once a sketch is trained, it allows for an efficient result size estimation of SQL queries. 105
Figure 47 Web interface for Deep Sketches. 106

LIST OF TABLES

Table 1 Comparison of different stream processing approaches. . 18
Table 2 Schema snippet of the Analytics Matrix. 21
Table 3 RTA queries 22
Table 4 Options to mitigate the network bottleneck depending on whether we control the clients and whether they run on the same machine as the database. 27
Table 5 Evaluated systems. 32
Table 6 Metrics of the NYC polygon datasets. 64
Table 7 Metrics of three super coverings with various precisions. 66
Table 8 Metrics of the different data structures (4 m precision). . 66
Table 9 Speedups of lookups in smaller (more coarse-grained) over larger (more fine-grained) polygon datasets for different data structures (b = boroughs, n = neighborhoods, c = census). 68
Table 10 Distribution of the tree traversal depth (ACT4 with 4 m precision). 70
Table 11 Performance counters per point (neighborhoods, 4 m precision). 71
Table 12 Speedups of single-threaded lookups when training ACT4 with an increasing number of historical data points (over untrained ACT4). 75
Table 13 Effect of training the index with 1 M historical data points (STH = solely true hits). 75
Table 14 Distribution of joins. 86
Table 15 Estimation errors on the synthetic workload. 87
Table 16 Estimation errors of 376 base table queries with empty samples in the synthetic workload. 88
Table 17 Estimation errors on the JOB-light workload. 90
Table 18 Cardinalities (distinct value counts) of columns used in our workload. Checkmarks indicate whether columns can appear in filter and/or group-by clauses. 97
Table 19 Estimation errors on the query workload. 99

1 INTRODUCTION

1.1 ANALYTICAL DATABASE SYSTEMS

Already back in 2006, Clive Humby coined the phrase *data is the new oil* [85]. Like oil, data also first needs to be processed to provide actual value. However, in contrast to oil, data is far more complex to turn into profit and requires sophisticated tools to extract actionable insights [66].

To unlock the potential of their data, companies today rely upon modern analytical database systems. The trend is moving away from proprietary or on-premise solutions to cloud offerings such as Amazon Redshift [75], Google BigQuery [22], and Snowflake [46].

These data warehouse systems offer many benefits for customers, including automatic maintenance, high reliability, and most notably *elasticity*. All of these may result in substantial cost savings. Likewise, the cloud opens up very interesting opportunities for database vendors. With high-level insight into user workloads, vendors can optimize their systems based on usage patterns.

1.1.1 Challenges

Modern analytical database systems are faced with various challenges due to evolving use cases and growing data volumes. Despite decades of research, core problems like query optimization and multi-dimensional indexing remain to a large extent unsolved with a lot of potential for further improvements.

Fresh Data

While these systems achieve state-of-the-art query performance, especially when considering cost, they by design operate on *stale* data. The reason is that they optimize for the *read-mostly* case and use techniques such as columnar storage, compression, and even reorganize entire tables to find sort orders favoring query patterns [14].

Thus, these systems are typically not suited to store the mission-critical transactional data of a company (e.g., customer orders). This data therefore remains in separate transaction-processing systems and is either continuously or periodically extracted into data warehouses [2, 165].

There are, however, applications where *time to insight* matters and where analytical queries need to run on the *most recent* state to deliver fresh results [30]. In

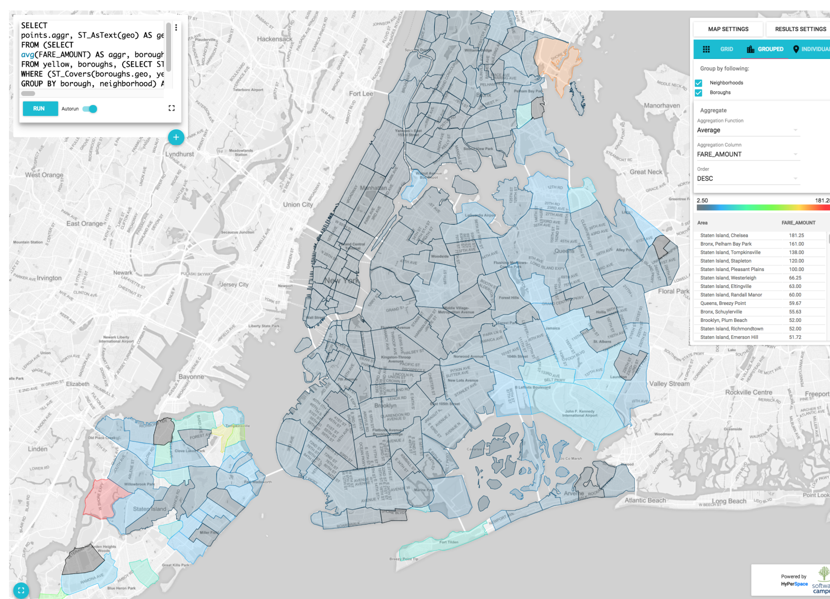


Figure 1: HyPerMaps. Interactive exploration of changing point datasets.

these cases, there is no time for transferring data between transactional (OLTP) and analytical (OLAP) systems. This need has created an entire new field in database systems research, and so called hybrid transactional/analytical processing (HTAP) systems have emerged. The main-memory database systems (MMDBs) HyPer [96] and SAP HANA [63] pioneered supporting both OLTP and OLAP workloads on the same database state. While they can omit the ETL process altogether, they only offer limited scalability in terms of data size and transaction throughput.

At the same time, modern streaming systems such as Apache Flink [32] have been developed to minimize time to insight for the specific use case of analyzing enormous amounts of event data. While they address the data freshness problem and provide better scalability than MMDBs, they typically do not implement ACID (Atomicity, Consistency, Isolation, Durability) guarantees [78]. Furthermore, event data remains separated from transactional data, prohibiting efficient queries across the two. In Chapter 2, we study how event data can be processed and simultaneously analyzed in MMDBs using materialized views and a scale-out architecture.

Geospatial Data

Driven by the shift to *mobile* a large portion of the data generated today comes with location information. The specific use cases are endless, and range from online processing of fresh data to interactive exploration of historical data. For example, the ride-hailing company Uber needs to join each passenger request with a set of predefined boundaries (polygons) to display available products (e.g., Uber X) and to enable dynamic pricing [203] while it processes

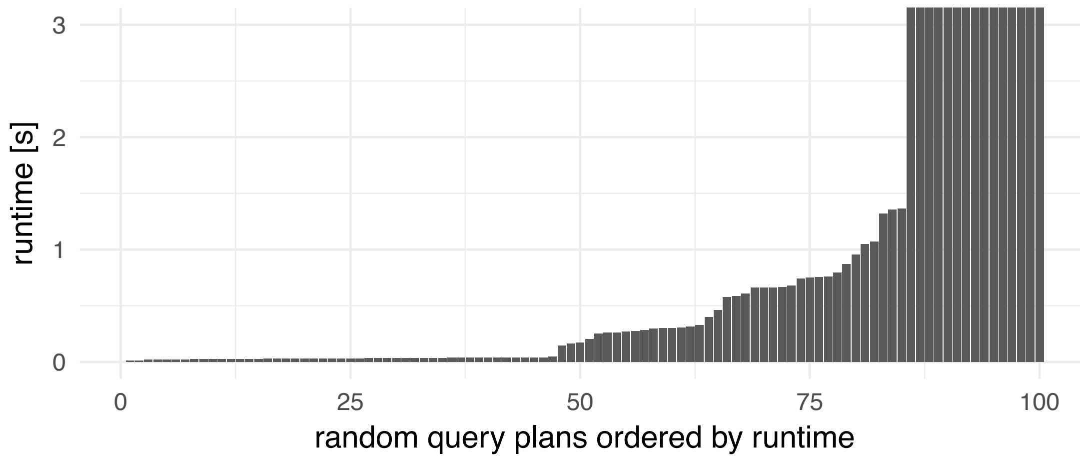


Figure 2: Runtimes of TPC-H Query 5 (SF1) with random query plans. Plot adapted from [152]. Numbers reproduced on AMD Ryzen Threadripper 1950X.

offline data to analyze usage patterns. Other use cases include location-based ads targeting and traffic-aware routing. Consequently, analytical databases are adding indexing and query processing support for geospatial data [13].

In earlier work we have extended the in-memory database HyPer [96] with geospatial data types and operators [163]. Figure 1 shows the corresponding web interface (HyPerMaps) with locations from the New York City (NYC) taxi dataset [196] aggregated by neighborhood. However, we found that to allow for interactive analytics on datasets like this, an efficient execution engine alone is not enough. Given the multi-dimensional nature of geospatial data, queries on such data can be very expensive [18]. It is also not straightforward to build good indexing support for such queries. There are many variants in literature, many of which were invented decades ago with different objectives in mind (e.g., disk I/O being the limiting factor) [87]. Hence, spatial indexing is still under active research with recent proposals focusing on utilizing modern hardware, including accelerators such as GPUs [215]. In Chapter 3, we revisit an approach called true hit filtering [31] and adapt it to modern hardware.

Query Optimization

Another open challenge in analytical databases is query optimization. Figure 2 shows the runtimes of 100 executions of TPC-H Query 5 (SF1) with random join orders [152] in HyPer [96] (the y-axis is cut off at 3 seconds). This query has five joins and its execution time varies from 10 milliseconds up to more than 10 minutes depending on the query plan. This demonstrates that query optimization is crucially important, and that even a fast runtime system like HyPer's JIT-compiling execution engine [151] cannot compensate for mistakes made by the optimizer.

With growing data volumes, this effect is exaggerated. While queries on small datasets may still complete with bad query plans (as in the above example), they may not finish on large datasets at all due to huge intermediate results. Particularly, in a distributed setting, choosing a bad plan can lead to disastrous performance due to the cost of shuffling data over the network. Techniques such as sideways information passing can mitigate the impact of bad join orders to some degree. The idea is to build Bloom filters [24, 119] on the build-side input of joins and pass these filters to the probe-side scan nodes for early data pruning [136]. Nonetheless, building and passing these filters along is also not for free and is limited to equi-joins. There is, however, a recent proposal on succinct range filters [223], which could be used for the same purpose. Likewise, changing a plan mid-query is difficult and expensive, particularly in compiling [151] and in distributed query engines [181]. Procella [36] mitigates this problem by only reoptimizing the parts of the plan that have not been executed. It collects data statistics in early stages of the execution, which are used to optimize later stages. But the problem remains, optimizer mistakes can have a significant performance impact.

Leis et al. showed that the core problem in cost-based query optimization is bad cardinality estimates and that widely-used optimizers are sometimes wrong by orders of magnitude [124, 128]. The same work also showed that correlations found in real-world data are causing this effect. In Chapter 4, we propose using deep learning to capture the correlations found in data to improve cardinality estimation.

1.1.2 Research Opportunities

Thanks to advances in hardware, high-level insights into user workloads in the cloud, and breakthroughs in machine learning, there are many interesting future research directions.

Large Main-Memory Capacities

DRAM capacities have increased substantially in the last decade. Not surprisingly, cloud vendors offer machines with increasing memory capacity. For example, AWS recently launched instances with up to 24 TiB of main memory [19] to run in-memory systems such as SAP HANA [63]. This allows latency-critical applications, such as in finance, to run purely in main memory. Even if the DRAM capacity of a single machine does not suffice, modern networking technologies such as InfiniBand allow to aggregate the main memory of an entire cluster with manageable latency impact [23].

However, many database algorithms and data structures were designed in times where main memory was scarce. Thus, a current trend is to revisit established techniques, such as buffer managers, and adapt them to modern hard-

ware [125]. In general, main memory can be leveraged in multiple ways. One is to avoid unnecessary I/O operations in disk-based systems by maintaining filters, such as Bloom filters [24], in main memory. This idea has already been explored in systems [49, 54], but we argue that we are still at the beginning and that data pruning is a widely open research topic. Another way is to maintain more fine-grained (i.e., selective) index structures to avoid unnecessary computations in in-memory systems. In other words, one can precompute partial results and thus trade off memory consumption with performance. In Chapter 3, we explore building a fine-grained in-memory radix index to accelerate the traditionally compute-intensive operation of geospatial joins.

Given the clear benefits of in-memory computing (i.e., low latency, high throughput), another interesting research direction is to fully store and index *compressed* datasets in main memory that would otherwise exceed main-memory capacity. Examples include the use of *succinct* data structures such as the Fast Succinct Trie (FST) [223] that manage to significantly compress data while being almost as query-efficient as state-of-the-art index structures such as the Adaptive Radix Tree (ART) [126]. The drawback of such approaches is their lack of *updatability*. Although, there is a proposal to combine read-optimized with updatable indexes [222], which could also be applied to succinct data structures.

Besides DRAM, another trend is to leverage byte-addressable non-volatile memory (NVM) in databases [174, 175, 10]. NVM offers more space than DRAM at a lower price point. While having higher latencies than DRAM, NVM is still an order of magnitude faster than flash drives [58]. Once commonly available, NVM is expected to fundamentally change the architecture of database systems and render many components that deal with potential memory loss as unnecessary, to the point where they will degrade performance [10].

Approximate Query Processing

To deal with growing data volumes, an attractive approach is to relax *precision* in query processing. Approximate results can potentially be computed orders of magnitude faster, allowing for interactive analytics on large datasets. Specifically, approximate query processing (AQP) allows for identifying data subsets that need further drill-down, trend discovery, and data visualization (e.g., heat maps) [38]. Concrete techniques include data summaries such as quantile sketches [68, 142], approximate top-k algorithms [145], and sampling-based approaches [70, 37, 208]. Recently, machine learning has been applied to AQP showing benefits over sampling [115, 202]. However, despite the clear benefits of AQP, it has still not been widely adopted. Chaudhuri et al. [38] argue that to make AQP practical, we need to allow users to control the precision of query results. In a recent study covering sample-based approximations at Microsoft, Kandula et al. also argue that the lack of accuracy guarantees is an

adoption barrier [92]. Thus, to make AQP a widely-used technology, we need to think about what controls to expose to users. A good example is the approximate top-k algorithm *Space-Saving* [145], which guarantees a user-defined error.

Likewise, many data are approximate in nature such as log entries that capture performance events or GPS locations obtained by smartphones [51]. Even if a query would process all data items, the result remains approximate [92]. In Chapter 3, we apply AQP to the problem of joining geospatial points with polygons by allowing users to control the *maximum distance* of false positive join partners.

Data-Driven Optimizations

Another interesting research direction is to *adapt* systems to user data and workloads. The idea is to make a system either periodically or continuously improve its efficiency over time by *specializing* it to concrete data and query patterns. Efficiency in this context can mean improving query performance but also reducing storage cost. An example is *database cracking* [86], in which the physical layout of a database is adapted one user query at a time such that popular data is faster to access. In Chapter 3, we use a similar approach to improve the effectiveness of a polygon index. Other examples that also made it into production systems include automatic clustering and sorting [12, 14]. Pavlo et al. [164] argue that we should take this even further and aim for a fully autonomous ("self-driving") database system that predicts future access patterns and prepares itself accordingly (e.g., adjusts its physical design).

With the recent advances in machine learning and hardware, we can explore new data-driven optimizations and adapt our systems to more complex patterns. We will cover recent proposals in this area in the following section.

1.2 THE LEARNED SYSTEMS ERA

Since Kraska et al. proposed to *learn* index structures [110], a groundswell of research has been dedicated to improving database systems with machine learning. Before diving into concrete examples, we will give some background on the advent of machine learning.

Machine learning (ML) is a branch of artificial intelligence and has had several breakthroughs in the last decade. Particularly deep learning, a family of ML methods based on neural networks, has led to significant advances in fields such as computer vision [113] and natural language processing [50]. In general, ML can be divided into the sub areas supervised, unsupervised (or self-supervised), and reinforcement learning. In supervised learning a model is trained with *labeled* training data, while in self-supervised learning a model

Figure 3: GFLOPS (32-bit) of various GPUs. Data collected by [182].

is trained with *unlabeled* data and learns structures embedded in the data. In reinforcement learning, a model learns from interactions with its environment and tries to maximize a *reward*.

Many of the recent breakthroughs are enabled by advances in hardware. Figure 3 shows the (peak) floating point performance of various (including consumer-class) graphics processing units (GPUs) over the last decade with a significant increase in recent years [182]. Even more notable is that the performance to power ratio (i.e., GFLOPS per watt) has improved dramatically. In particular, Tensor Processing Units (TPUs) report a >30× improvement over an NVIDIA K80 GPU in that respect [90]. However, not all deep learning algorithms are compute bound, some are actually bound by memory latencies and bandwidth [130]. But even there, we have seen tremendous improvements with recent GPUs such as the NVIDIA Tesla V100S offering 1134 GB/s of memory bandwidth [158]. Furthermore, the bandwidth between CPUs and dedicated GPUs is improving with interconnects like NVIDIA NVLink 2.0 promising up to 300 GB/s of aggregated bandwidth [157].

Thus, using machine learning to improve systems is more attractive than ever, and will be even more accessible once such compute units become a commodity. Not surprisingly, we have seen many proposals of using ML to improve systems. Kraska et al. envision to enhance or even replace tradi-

tional database components with ML counterparts to build a learned database system [109]. Recent work proposes to use ML for classical [110, 103] and succinct [209] index structures, view materialization [132], index tuning [11, 52], data partitioning [83], workload management [88], and query performance prediction [139]. Most notably, and related to this thesis, there have been many proposals in the query optimization space. Similar to IBM's learning optimizer LEO [187] which learns from past optimizer mistakes, SkinnerDB [198] does so during the execution of a single query using reinforcement learning. Other proposals focus on join enumeration [140, 112], plan rewriting [47], and even describe an end-to-end learned optimizer [138]. Given the importance of cardinality estimation for cost-based query optimization [124], it is also not surprising that many proposals have been made in that respect [161, 206, 211, 57, 205]. This includes our own supervised learning approach, which is a core contribution of this thesis and is described in Chapter 4.

1.3 CONTRIBUTIONS

This thesis addresses the challenges and opportunities described above and contributes to the research area of analytical database systems as follows:

ANALYTICS ON FAST DATA Michael Stonebreaker, who received the prestigious Turing Award, once identified eight rules [188] that real-time stream processing engines should follow, including the support for SQL as a query language and the integration of stored and streamed data. However, up to date, these requirements have not been fully addressed. For example, modern streaming systems such as Apache Flink [32] do not allow to query globally consistent state (e.g., aggregates) without transferring data to external systems. On the other hand, main-memory database systems (MMDBs) such as HyPer [96] achieve an unprecedented SQL query performance and allow for concurrent updates.

In Chapter 2, we study how MMDBs perform in analyzing (aggregated) event stream data. Based on a benchmark proposed by Braun et al. [30], we define a new class for streaming workloads called *analytics on fast data*. These workloads aggregate event data into materialized views that are concurrently analyzed with ad-hoc SQL queries. These queries come with *data freshness* requirements such that it is not feasible to transfer data to external analytical systems. We compare MMDBs to modern streaming systems in this context and identify issues in both, including lack of support for streaming SQL in databases as well as inefficient access to materialized state in streaming systems. Based on our analysis, we derive and implement extensions to MMDBs to better support *analytics on fast data*, and to match the performance and usability of streaming systems. Our extensions include the use of user-space

networking to mitigate the network bottleneck in these systems, a study on how MVCC and optimistic locking allows for improved concurrency, and a scale-out architecture.

GEOSPATIAL ANALYTICS As stated earlier, a large portion of data generated today is location aware and the *spatial join* is arguably the most expensive operation in processing such data. Given that many approaches to accelerate this operation were developed in a different time under different objectives, we argue that it is time to revisit the seminal work in this area [87, 159, 31, 191] and adapt it to modern hardware.

In Chapter 3, we propose a novel radix tree-based polygon index to accelerate point-polygon joins in main memory. We first propose an approximate variant that trades off precision with performance and allows users to control the *maximum* error. Compared to other AQP approaches in the spatial domain [190, 215], ours uses a hierarchical grid (an implicit quadtree) to approximate polygons. To our knowledge, this is the first work to provide a distance-based precision bound in a hierarchical polygon index. For a join between NYC's yellow taxi data and NYC's neighborhood polygons, it achieves more than 50 M point lookups/s per CPU core under a <4 m precision bound. For cases when approximate results are not sufficient or when main memory is scarce, we also offer an exact variant. This variant makes extensive use of *true hit filtering* [31] to identify *most* join partners during the index lookup, avoiding expensive geometric computations. Most notably, it allows for *training* the index with historical query points to become more effective in popular areas.

LEARNED CARDINALITIES We lastly shift our focus from query processing to *optimization*. As we have motivated earlier, the query optimizer is a critical component in an analytical database system. Mistakes made by the optimizer can hardly be compensated by the execution engine. With the recent breakthroughs in ML and advances in hardware, it is now feasible to efficiently *learn* parts of the optimizer. Some recent proposals to use ML in this context focus on join enumeration [140, 112]. However, given that modern join enumeration algorithms can find the *optimal* join order for queries with dozens of relations [155], we argue that we should rather focus on the core problem in cost-based query optimization: *cardinality estimation*. As Guy Lohman once stated, cardinality estimation is the "Achilles heel" of query optimization [135] and causes most of its performance issues [124]. Given the highly non-trivial nature of the cardinality estimation problem and the poor performance of current implementations [124], we believe that ML can make a large difference.

In Chapter 4, we propose a new deep learning approach to cardinality estimation. Our contribution is the use of a novel *set-based* model, which is based on the insight that the cardinality of a query is independent of the concrete query plan. Also, and most notably, our approach integrates *runtime sampling*.

Specifically, we use bitmaps that indicate qualifying base table samples as an additional input signal to the model. This is in contrast to related work that uses ML without incorporating any runtime features [161, 206]. Compared to pure sampling, our approach mitigates the impact of 0-tuple situations (i.e., no qualifying samples) by still being able to rely on query features in such cases. We apply our model to two important cardinality estimation problems: (i) join size estimation and (ii) estimating the result sizes of group-by queries in the presence of selections. In both cases, our model achieves state-of-the-art performance on a real-world dataset. We also demonstrate the end-to-end training and inference process and sketch a possible optimizer integration.

2 ANALYTICS ON FAST DATA

Excerpts of this chapter have been published in [104, 105].

2.1 INTRODUCTION

Gartner recently forecasted that there will be more than *20 billion* connected devices in 2020, a 400% increase compared to 2016 [72]. The growing popularity of Internet of Things applications [143], including connected vehicles, cell phones, and health monitoring devices, enable a variety of new business use cases and applications. These applications are typically built around streaming systems that are able to ingest and aggregate enormous amounts of events from different data sources. Given the spike of interest in building such applications, it is not surprising that dedicated stream processing systems, like Apache Storm [197], Apache Spark Streaming [217], or Apache Flink [32], are receiving significant attention not only in the database but also in the data science and in the open-source community.

To better understand the different types of workloads that these systems need to handle, we will walk through different ways of processing sensor readings (events) of connected vehicles that contain information about street conditions such as icy road segments.

First, a streaming system could warn vehicles about icy road segments based on the information of single events. In that case, the streaming system does not need to maintain state. We refer to such workloads as *stateless* streaming.

Second, a system could process the aggregated information of multiple events to decide whether vehicles should be warned. Such an implementation requires the system to maintain a processing state, which introduces new challenges such as consistency and durability. We call this kind of workloads *stateful* streaming.

Third, a streaming system allows users to perform analytical queries on the entire set of aggregates (the conditions of all road segments across the city) to find the most critical segments. We refer to such workloads as *analytics on fast data*. These workloads are particularly challenging for a streaming system since it needs to perform computations across multiple partitions[1] to answer analytical queries (cf. Figure 4). In other words, the problem here is how to efficiently aggregate streaming events, store and maintain these aggregates,

1 When referring to partitions, we mean horizontal partitions of the global state.

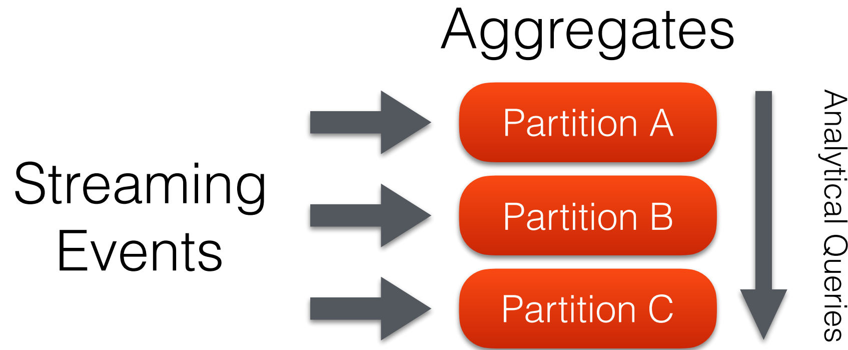


Figure 4: Analytics on fast data. Streaming events may be processed concurrently in different partitions, whereas analytical queries cross partition boundaries and require a consistent state.

and query the whole set of aggregates. In fact, without modifications, none of the streaming systems mentioned above can handle this use case. The reason for this is that these systems either do not expose global state to analytical queries at all or only support point lookups (cf. Section 2.3.2). One idea to mitigate this problem is to make use of the fact that these systems periodically flush their state to durable storage (e.g., HDFS) to address fault tolerance. This means that the system state becomes queryable for an analytical engine like Apache Spark [216]. However, the delay that this design introduces prohibits analytical queries to run on the most recent state, which is required by use cases like the one above.

Another example is the Huawei-AIM telecommunication workload [30]. In this use case, events represent sales and marketing information generated by phone calls. On the one hand, the application needs to maintain a huge set of aggregates per customer in order to trigger alerts for a particular customer (*stateful* streaming). On the other hand, maintenance specialists might query the overall system state to localize sources for network failures or business analysts might run analytics to gather insights and propose new offers in real time (*analytics on fast data*). Again, using an off-the-shelf stream processor does not solve this use case since it cannot handle the real-time analytics. There are, however, state-of-the-art main-memory database systems (MMDBs) dedicated to handle mixed OLTP and OLAP workloads, such as HyPer [96] (cf. Section 2.2.1) and Tell [167] (cf. Section 2.2.1), which seem promising because stream processing could also be seen as a particular class of OLTP workloads. These systems feature advanced query optimizers, compile queries to native code, and can thus achieve extremely low response times for complex analytical queries. Using efficient snapshotting mechanisms, such as *copy-on-write*, *MVCC*, or *differential updates* [96, 154, 114], these systems are able to sustain high transaction throughput rates in parallel to analytical query processing making them well-suited for workloads where analytical queries need to consider recently ingested data.

Despite all these advantages, it seems that data engineers are still reluctant to use MMDBs for stream processing. They either build their own solutions on top of modern streaming systems (e.g., Apache Flink) or hand-craft systems from scratch that are specifically tuned for particular workloads (e.g., AIM [30]). One reason that MMDBs are not widely used for streaming workloads is that they lack out-of-the-box streaming functionality, such as window functions, and adding this functionality (e.g., through stored procedures or user-defined functions) results in additional engineering. If MMDBs would offer better support for streaming workloads (e.g., streaming extensions for SQL as proposed in StreamSQL [188]), they would be preferable over hand-crafted systems, which are also costly to maintain.

In this work, we thoroughly evaluate the usability and performance of database systems, modern streaming systems, and AIM, a hand-crafted system, using the Huawei-AIM workload [30]. Based on the evaluation results, we answer the question how off-the-shelf MMDBs can be extended to sufficiently satisfy the requirements of *analytics on fast data*. We identify a set of modifications that, if properly applied to the off-the-shelf MMDBs, allow these systems to address the needs of *analytics on fast data*. In this extended version of our previous work [104], we have started to close the gap between MMDBs and modern streaming systems:

- We discuss how to address the network bottleneck, which is a significant issue in update-heavy workloads (cf. Section 2.4.1).
- We outline how event processing throughput can be scaled within a machine and discuss parallel transactions, lightweight synchronization using optimistic locking, and skew handling strategies (cf. Section 2.4.2).
- We demonstrate how MMDBs can be scaled to multiple machines (cf. Section 2.4.3).
- We show how optimistic locking allows for parallelizing (single-row) transactions in MMDBs (cf. Section 2.5.4).
- We have introduced skew into the Huawei-AIM workload and discuss the consequences in Section 2.5.8.
- We have designed a microbenchmark that evaluates the impact of user-space networking (cf. Section 2.5.9).
- And finally, we have evaluated the systems in a distributed setting (cf. Section 2.5.10).

In summary, our contributions include:

- A rich survey of various MMDBs, modern streaming systems, and a hand-crafted system specifically designed to address the Huawei-AIM workload

- A thorough usability and performance evaluation including at least one representative of each of these classes of systems
- A discussion of how MMDBs can be extended to match the performance and usability of modern streaming systems
- A discussion and evaluation of concrete extensions to MMDBs that increase their streaming capabilities

The remainder of this chapter is structured as follows: Section 2.2 summarizes a broad variety of existing systems and Section 2.3 revisits the Huawei-AIM workload and describes how it can be implemented with these systems. Section 2.4 discusses concrete extensions to MMDBs that increase their streaming capabilities. Section 2.5 evaluates the performance of representatives of each kind of system with respect to this workload. Section 2.6 enumerates ideas regarding how to further close the performance and usability gap between MMDBs and modern streaming systems and is followed by the conclusions to our evaluation presented in Section 2.7.

2.2 APPROACHES

There are numerous systems that can be used to build stream processing pipelines, including near real-time data warehousing solutions like Mesa [76] and in-memory incremental analytical engines like Trill [34]. S-Store [144] is an approach to integrating stream processing into an OLTP engine. Since addressing all of these systems is beyond the scope of this work, we will focus on representative MMDBs, popular streaming systems from the open-source domain, and AIM [30], a hand-crafted, highly-optimized solution.

2.2.1 Main-Memory Database Systems

There are multiple MMDBs that can handle *analytics on fast data* or more generally hybrid transactional/analytical processing (HTAP) workloads. In HTAP, transactions are usually more complex (e.g., TPC-C transactions) than the single-row transactions studied in this work.

HyPer

HyPer[2] is a MMDB that achieves an outstanding performance for both OLTP and OLAP workloads, even when they operate simultaneously on the same

2 When saying HyPer, we are referring to the research version of HyPer developed at the Technical University of Munich.

database [96]. HyPer versions individual attributes using *multi version concurrency control* (*MVCC*) to isolate concurrent transactions [154]. However, HyPer currently avoids the cost of synchronizing the data access. Instead, write transactions and analytical queries are interleaved and do not run simultaneously. HyPer further features data-centric LLVM code generation with just-in-time compilation. Finally, HyPer has an advanced dynamic programming-based optimizer including the ability to unnest arbitrary queries [153].

Tell

Tell [167] is a distributed shared-data MMDB that supports OLTP and OLAP in parallel and is developed at the Systems Group at ETH Zurich. The implementation of Tell is fundamentally different from that of other systems presented in this work as it separates the computation from the storage layer in such a way that both layers can scale out individually [134].

The storage layer, TellStore, is a versioned key-value store with additional support for fast scans and different storage layout options, such as *RowStore* and *ColumnMap*. *ColumnMap*, the preferred layout for HTAP workloads, was created as part of Analytics in Motion (AIM) [30] (cf. Section 2.2.3) and is a modified *Partition Attributes Across* (PAX) [3] approach that optimizes cache locality by storing data column-wise in blocks of cache size. This optimization allows *ColumnMap* to support fast scans and, at the same time, reasonably fast record lookups and updates. TellStore employs the *shared scan* technique, which allows incoming scan requests to be batched and processed all at once by a single thread. The *shared scan* can be parallelized efficiently by partitioning the data and using a dedicated scan thread for each of these partitions in parallel [200]. Isolation is guaranteed using a combination of *differential updates* [114] and *MVCC*. Updates are put into a *delta* data structure, which gets periodically merged with the *main* data structure that serves analytical queries. This approach is also used in SAP HANA [63].

Tell's compute layer offers two processing APIs: TellDB (C++) for general-purpose transactions and TellJava (Java) for read-only analytics. TellJava can be further integrated into distributed processing frameworks, including Apache Spark and Presto.

2.2.2 Modern Streaming Systems

In addition to MMDBs, there are dedicated streaming systems allowing for the implementation of streaming pipelines. These systems provide out-of-the-box functionality, including a rich set of operators to help data engineers to address the specific demands of streaming use cases.

Apache Samza

Apache Samza [156] is a distributed framework for continuous real-time data processing that is lightweight, elastic, and fault-tolerant. Samza uses Apache Kafka [111] (a durable publish-subscribe-based message passing system that allows replaying messages) for real-time feeds and produces output feeds for Kafka to consume. For distributed scheduling, fault tolerance, and resource allocation, Samza depends on Apache YARN and on Kafka. Samza employs a checkpointing mechanism to provide *at-least-once* guarantees. It creates checkpoints at predefined time intervals and in case of a job failure, it replays messages from the last checkpoint. A drawback of Samza is that it does not support *exactly-once* semantics. A message might be processed twice after a job failure, which can lead to non-exact results. That effect can be minimized by using shorter checkpoint time intervals.

Apache Flink

Apache Flink [32] is a combined batch and streaming processing system that supports *exactly-once* semantics. Flink follows a *tuple-at-a-time* approach, providing low latency. Using asynchronous checkpointing, Flink is able to decouple its fault-tolerance mechanism from the tuple processing. The processing continues while Flink periodically creates snapshots of the operator states and the in-flight tuples. Flink can achieve superior throughput compared to Apache Storm (cf. Section 2.2.2). In contrast to the other streaming systems, Flink allows for event time semantics. Flink allows the extraction of the actual event timestamp (i.e., the time when the event was originally captured) when an event arrives at the streaming engine to assign it to its appropriate window. Starting with version 1.2.0, Flink supports a global queryable state. The idea is to maintain an operator-independent state within Flink and expose it to external queries. Internally, the state is partitioned and guarantees fault tolerance (i.e., *exactly-once* semantics). A restriction of this solution is that it is only a key-value state supporting only point lookups. More complex queries, including full table scans, are not possible.

As a workaround, we could implement a custom operator that holds both the state and the logic for the corresponding analytical queries. The drawback of this approach is that the whole state and query logic has to be implemented manually. Further, this approach does not support concurrent stream and query processing since analytical queries can only be ingested through the stream processing pipeline itself resulting in an interleaved execution.

Apache Spark Streaming

Apache Spark Streaming [217] is the streaming extension to the cluster computing platform Apache Spark. Spark Streaming organizes incoming stream-

ing tuples into *micro-batches* that are being processed atomically. Processing batches of tuples increases the throughput on the cost of higher latencies. Batching tasks also eases load balancing and recovery from failures. Tasks can be distributed dynamically on the cluster and failed tasks can be relaunched on any machine. Another advantage of this approach is that it allows the use of the same programming model for batch and stream processing. Spark Streaming supports *exactly-once* semantics.

Apache Storm

Apache Storm [197] is a widely used stream processing system that does not guarantee state consistency and follows a *tuple-at-a-time* approach, thus favoring low latency over throughput. Storm implements *at-least-once* semantics by keeping upstream backups of data that are being replayed if no acknowledgements have been received from downstream nodes. Trident [9] extends Storm with *exactly-once* semantics and allows running queries on consistent state.

2.2.3 AIM

In collaboration with Huawei, researchers of the Systems Group at ETH Zurich designed the AIM system to address the specific characteristics of a telecommunications workload. AIM is a research prototype that allows efficient aggregation of high-throughput data streams. It was specifically designed to address the Huawei-AIM workload that we use for evaluation purposes in this work (cf. Section 2.3). Due to its hand-optimized nature, AIM achieves an outstanding performance on that workload and therefore serves as a baseline for our experiments. AIM has a three-tier architecture consisting of storage, event stream processing (ESP), and real-time analytics (RTA) nodes (or threads if deployed in a standalone setting). RTA nodes push analytical queries down to the storage nodes, merge the partial results, and finally deliver the results to the client. ESP nodes process the incoming event stream and update corresponding records by sending *Get* and *Put* requests to the storage nodes. The storage nodes store horizontally-partitioned data in a *ColumnMap* layout and employ *shared scans* as described in Section 2.2.1. AIM can also be deployed standalone, which eliminates network costs and therefore tests the pure read, write, and scan performance of the server.

2.2.4 Summary

A comparison of different aspects of stream processing approaches is presented in Table 1. These aspects include:

SEMANTICS Streaming engines make different guarantees regarding how messages (i.e., events) are being processed. A streaming engine only ensures

Table 1: Comparison of different stream processing approaches.

	MMDBs		Modern Streaming Systems				
Aspect	**HyPer**	**Tell**	**Samza**	**Flink**	**Spark Streaming**	**Storm**	**AIM**
Semantics	Exactly-once	Exactly-once	At-least-once	Exactly-once	Exactly-once	Exactly-once	Exactly-once
Durability	Yes	No	With durable data source	With durable data source	With durable data source	With durable data source	No
Latency	Low	Low	High (writes messages to disk)	Low	Medium (depends on batch size)	Low	Low
Computation model	Tuple-at-a-time	Tuple-at-a-time	Tuple-at-a-time	Tuple-at-a-time	Micro-batch	Micro-batch	Tuple-at-a-time
Throughput	High	High	High	High	Medium (depends on batch size)	Low	High
State management	Yes	Yes	Yes (durable K/V store)	Yes	Yes (writes into storage)	Yes	Yes
Parallel read/write access to state	MVCC	Differential updates, MVCC	No	No	No	No	Differential updates
Implementation languages	C++, LLVM	C++, LLVM	Java, Scala	Java	Java, Scala	Java, Clojure	C++
User-facing languages	SQL	C++, Java, Scala (through Spark shell), SQL (through Presto shell)	Java, Scala	Java, Scala, SQL (through Apache Calcite)	Java, Scala, Python, SparkSQL	Any (through Apache Thrift)	C++
Own memory management	Yes	Yes (w/ GC)	No	Yes	Yes	No	Yes
Window support	Using stored procedures	Only manually	Very basic	Very powerful	Basic	Basic	Using template code

completely correct results when providing *exactly-once* guarantees. Some engines optimize for low latency and thus often cannot provide *exactly-once* guarantees as this would require them to implement transactions, which are expensive in a distributed setting. Therefore, streaming engines often fall back to *at-least-once* semantics (i.e., a message will be re-sent until it is processed at least once), which are good enough for many applications. Many stream processing engines require a durable data source for *exactly-once* guarantees because they only persist their process-

ing state at certain points of time (often called checkpoints). In case of a failure, messages need to be replayed from the last checkpoint. In contrast, database systems achieve durability through the use of redo logs and thus only need to replay messages sent during the time the database system was down. The third processing guarantee is *at-most-once*. In an *at-most-once* setting, messages might get lost but are never processed twice or more often. Few systems implement this approach since loosing data is an undesirable property for most applications.

DURABILITY Durability is closely related to the semantics offered by stream processing systems. While some systems require a durable data source to achieve durability, others provide durability out-of-the-box.

LATENCY Especially in real-time scenarios, low latencies are crucial to deliver valuable results. As stated above, latency often depends on the processing guarantee offered by a system. MMDBs that often run on a single machine or are optimized for low-latency networks can yield low latencies while providing *exactly-once* processing guarantees.

COMPUTATION MODEL There are two computation models: *tuple-at-a-time* and *micro-batch*. The natural approach is to process streams continuously. However, streams can also be batched and processed as small chunks of data. Spark Streaming follows this approach allowing it to achieve high throughput rates. However, following a *tuple-at-a-time*-based approach does not necessarily lead to lower throughput since the computation model can be independent from the checkpointing interval. For instance, Flink follows a *tuple-at-a-time*-based approach combined with a batch-based checkpointing mechanism thus optimizing for both latency and throughput. MMDBs usually treat stream events as transactions, which might also be batched for better performance (e.g., Tell processes 100 events within a single transaction).

THROUGHPUT Another important aspect in stream processing is throughput. Particularly when costs matter, higher throughput helps to reduce the number of required resources. Due to the low costs to process single-row transactions (updating aggregates of single entities), throughput mainly depends on the employed fault-tolerance mechanism and whether a system batches transactions. Throughput increases with longer checkpointing intervals.

STATE MANAGEMENT For mixed OLTP and OLAP workloads, the state updated by the OLTP subsystem needs to be exposed to the OLAP subsystem. Traditional streaming engines, such as Apache Storm, do not allow maintaining state. They are only designed to process and transform an

input into an output data stream preventing writing *stateful* stream processing applications (e.g., aggregations over windows). Trident extends Storm with state management capabilities. Flink only maintains states on an operator basis and currently does not support global states that can be accessed by analytical queries. Database systems, on the other hand, can persist streaming results in temporary tables allowing OLAP queries to access them as if they were regular database tables.

PARALLEL READ/WRITE ACCESS TO STATE As mentioned earlier, Trident extends Storm with state management functionalities; however, it does not allow analytical queries and updates to access state in parallel. Instead, they have to be interleaved to ensure a consistent view of the state. In contrast, modern MMDBs can efficiently expose their current state to analytical queries through the use of snapshotting mechanisms, such as *copy-on-write*, *MVCC*, or *differential updates*.

IMPLEMENTATION LANGUAGES Most of the streaming systems are written in a JVM-based language, whereas MMDBs are usually implemented in C or C++. The trend is to compile queries to native code. HyPer and Tell use LLVM as a compiler backend.

USER-FACING LANGUAGES The Apache systems primarily support JVM-based languages while the MMDBs both support SQL and, in the case of Tell, additional languages through its Spark and Presto integration.

OWN MEMORY MANAGEMENT Whether a system employs its own memory management or fully relies on the memory management of the JVM. Spark Streaming and Flink are based on the JVM but still employ their own memory management to have a better control over garbage collection cycles.

WINDOW SUPPORT In streaming applications, aggregations are usually computed on a window basis. Two basic window types are sliding and tumbling. Sliding windows are contiguous time or count-based intervals, such as *last 24 hours* or *last 10,000 events*. Tumbling windows are non-overlapping time or count-based intervals, such as *today* or *every 10,000 events*. All of the analyzed streaming engines support these two kinds of windows. In particular, Flink offers extensive functionality to specify windows, supporting custom window assigners, triggers, and evictors. AIM supports tumbling windows for specific time intervals and the standard aggregation functions through templated code. The window definitions are loaded at startup and cannot be changed afterwards. The analyzed MMDBs have no natural window support. Some MMDBs allow to implement windows using stored procedures.

2.3 WORKLOAD

AIM was motivated by a telecommunication workload, which we will refer to as Huawei-AIM use case [30]. We choose this workload as it is well-defined and represents the workload class of *analytics on fast data*.

2.3.1 Description

The Huawei-AIM use case requires events, more specifically *call records*, to be aggregated and to be made available to analytical queries. The system's state, which AIM calls the *Analytics Matrix*, is a materialized view on a large number of aggregates for each individual subscriber (customer). There is an aggregate for each combination of aggregation function (min, max, sum), aggregation window (this day, this week, ...), and several event attributes as shown in Table 2, which shows a small part of the conceptual schema of an *Analytics Matrix*. For instance, there is an aggregate for the shortest duration of an international phone call today (attribute *min* in Table 2). The number of such aggregates (which defines the number of columns of the *Analytics Matrix*) is a workload parameter with default value 546. The *Analytics Matrix* also contains foreign keys to dimension tables. Since these dimension tables are very small, we omit them in our experiments.

Table 2: Schema snippet of the Analytics Matrix.

<table>
<tr><td rowspan="4">subscriber ID</td><td colspan="5">international calls</td><td>...</td></tr>
<tr><td colspan="4">today</td><td>...</td><td>...</td></tr>
<tr><td rowspan="2">count</td><td colspan="3">duration</td><td>...</td><td>...</td><td>...</td></tr>
<tr><td>sum</td><td>min</td><td>max</td><td>...</td><td>...</td><td>...</td></tr>
</table>

The use case requires two things to be done in real time: (a) update *Analytics Matrix* and (b) run analytical queries on the current state of the *Analytics Matrix*. (a) is referred to as Event Stream Processing (ESP) and (b) as Real-Time Analytics (RTA). When an event arrives in ESP, the corresponding record in the *Analytics Matrix* has to be atomically updated. RTA, on the other hand, is used to answer business intelligence questions. RTA queries are continuously being issued by one or multiple clients and are evaluated on a consistent state of the *Analytics Matrix*. This consistent state (or snapshot) is not allowed to be older than a certain bound t_{fresh}, which is a service level objective (SLO) of the Huawei-AIM benchmark and defaults to one second. Table 3 shows the seven queries from the original benchmark [30]. Additionally, users may issue ad-hoc queries. Since ad-hoc queries are not available upfront and can involve any number of attributes, it is impractical for a stream processing system to create specialized index structures.

Table 3: RTA queries 1 to 7, $\alpha \in [0,2]$, $\beta \in [2,5]$, $\gamma \in [2,10]$, $\delta \in [20,150]$, $t \in$ SubscriptionTypes, $cat \in$ Categories, $cty \in$ Countries, $v \in$ CellValueTypes

Query 1: SELECT AVG (total_duration_this_week) FROM AnalyticsMatrix WHERE number_of_local_calls_this_week > α;
Query 2: SELECT MAX (most_expensive_call_this_week) FROM AnalyticsMatrix WHERE total_number_of_calls_this_week > β;
Query 3: SELECT (SUM (total_cost_this_week)) / (SUM (total_duration_this_week)) as cost ratio FROM AnalyticsMatrix GROUP BY number_of_calls_this_week LIMIT 100;
Query 4: SELECT city, AVG(number_of_local_calls_this_week), SUM(total_duration_of_local_calls_this_week) FROM AnalyticsMatrix, RegionInfo WHERE number_of_local_calls_this_week > γ AND total_duration_of_local_calls_this_week > δ AND AnalyticsMatrix.zip = RegionInfo.zip GROUP BY city;
Query 5: SELECT region, SUM (total_cost_of_local_calls_this_week) as local, SUM (total_cost_of_long_distance_calls_this_week) as long_distance FROM AnalyticsMatrix a, SubscriptionType t, Category c, RegionInfo r WHERE t.type = *t* AND c.category = *cat*, AND a.subscription type = t.id AND a.category = c.id, AND a.zip = r.zip GROUP BY region;
Query 6: *report the entity-ids of the records with the longest call this day and this week for local and long distance calls for a specific country cty*
Query 7: SELECT (SUM (total_cost_this_week)) / (SUM (total_duration_this_week)) FROM AnalyticsMatrix WHERE CellValueType = *v*;

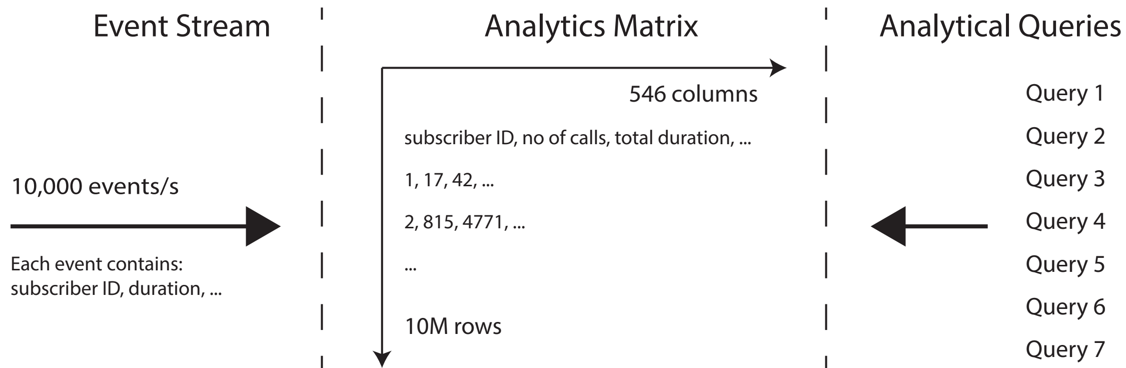


Figure 5: The Huawei-AIM workload.

Figure 5 summarizes the workload components. Events are ingested at a specific rate f_{ESP}, which will usually be 10,000 events per second in our experiments. Each event consists of a subscriber ID and call-dependent details, such as the call's duration, cost, and type (i.e., local or international). The *Analytics Matrix* is the aggregated state on the call records as described earlier and consists of 546 columns and 10 M rows, each representing the state of one subscriber. Depending on the event details, the corresponding subset of columns in the *Analytics Matrix* is updated for the particular subscriber. These updates are made available to analytical queries within t_{fresh}.

We use the Huawei-AIM-Simple-Standalone variant [29], which does not include the telecommunication industry specific advertisement campaigns from the original use case [30]. When we evaluate the systems in a distributed setting (cf. Section 2.5.10), we use the Huawei-AIM-Simple variant, which additionally involves network communication between clients and servers.

2.3.2 Implementations

We implement the workload in at least one representative of each of the three categories: MMDBs, modern streaming systems, and hand-crafted systems. We choose Flink as a representative modern streaming system since it features a continuous processing model combined with a fault-tolerance mechanism allowing for low latency under high throughput conditions. Flink's *tuple-at-a-time* approach gives more flexibility than the micro-batch model used by competitors such as Spark Streaming. For a performance comparison of Flink and Spark Streaming we refer the reader to related work [95, 42].

Since the Tell project (cf. Section 2.2.1) is no longer actively maintained, we had difficulties running new experiments on it and therefore do not further evaluate it in this extended version. We refer the reader to the initial publication of this work for a performance evaluation of Tell [104].

HyPer

Our workload implementation in HyPer is based on the work of [30]. ESP is performed using a pre-compiled stored procedure (written in SQL) that updates aggregates stored in the *Analytics Matrix*, which is implemented as a regular database table. RTA query processing is implemented using SQL queries on that table.

When HyPer was first evaluated using the Huawei-AIM benchmark in [30], HyPer was configured to use a *copy-on-write*-based snapshotting technique that forked a child from the main OLTP process at a specific time interval. This enables RTA queries to be executed on a consistent snapshot of the *Analytics Matrix*. Since the table representing the *Analytics Matrix* can be as large as 50 GiBs, forking a child of the OLTP process (essentially a copy of its page table) may take up to a hundred milliseconds. Additionally, our workload updates the records of randomly selected subscribers at a rate of 10,000 events/s, which may impact performance as the *copy-on-write* mechanism copies updated pages to maintain consistent snapshots for RTA queries.

In this work, we evaluate multiple versions of HyPer, a stable version (which we refer to as HyPer) that does not implement physical *MVCC*[3] as well as an experimental version HyPerParallel that parallelizes transactions (cf. Section 2.4.2) and a distributed variant of the stable version HyPerDistributed (cf. Section 2.4.3). All evaluated versions interleave the execution of multiple analytical queries to hide memory latencies and single-threaded phases (e.g., result materialization). Writes, however, are never executed at the same time than analytical queries.

HyPer implements the PostgreSQL wire protocol allowing one to use any PostgreSQL client. In our experiments, we use PostgreSQL's C++ library (pqxx) to communicate between clients and HyPer (using TCP over UNIX domain sockets). Since HyPer currently does not implement batched transactions, HyPer's event processing throughput would be purely limited by network round trips between subsequent write requests, context switches on the server to receive incoming requests, and deserialization costs. To simulate batch processing, we decided to additionally generate the events within HyPer and only process these. In other words, instead of actually transferring the batch of events from the client to the server, we send a request to generate and process a specified number of events.

AIM

Since the AIM system was specifically designed to address the Huawei-AIM workload, we assume that it would achieve the best performance on the full workload and thus we use it as a baseline for our experiments. We use the same

3 [154] explains how versioned positions allow for fast scans.

version used in [30] but in standalone mode where client and server communicate through shared memory. For the overall and the read-only experiments, we increase the number of RTA threads (and used one ESP thread), whereas for the write-only experiments, we increase the number of ESP threads.

Flink

A global state can be managed in Flink using a *queryable state*. A queryable state is a partitioned key-value store that allows users to perform queries from outside of the Flink job. However, this queryable state only supports point lookups and thus cannot be used to implement the AIM workload. We also could not use continuous queries since the idea of the Huawei-AIM workload is to query the state ad hoc. Continuous queries in Flink can be regarded as a stateful way of transforming one stream into another.

We have implemented a custom operator that supports table scans to meet the requirements of the AIM workload. We have experimented with a row and a column store layout for storing the state. Since the AIM workload is mostly analytical, we opt for the column store layout.

Similar to HyPer, we generate the events internally in Flink. We have also implemented a version that uses Kafka for event ingestion, which will not be included in the results, as we found no significant difference in performance compared to the version that generates the events internally. In production, Kafka, or any other durable data source, is preferable to ensure full fault tolerance.

Since we want to make the most recent state available to analytical queries, windows need to be computed on an event basis. As Flink's built-in operators are not optimized for these continuous window computations, we have manually implemented the window logic, which yields better results. We do not enable Flink's checkpointing mechanism since the processing state of the Huawei-AIM workload can be as large as 50 GiB. Persisting a state of this size would lead to a significant performance penalty.

Flink provides many built-in functionalities that seem suitable for our workload including windowed streams supporting various aggregation functions (e.g., min, max, and sum). We have tried to make use of the provided functionalities. However, in the studied version of Flink, combining multiple aggregation functions that produce only one single output stream is not yet supported. For this reason, we have implemented a custom aggregation operator.

All aggregations in the AIM workload are windowed. We could express this behavior using Flink's built-in window operators. For only one window type, this works well. However, with two or more different window types, the different windows would need to be merged into one consistent state across all windows. As this is not a straightforward operation in Flink, we have decided to implement windows ourselves.

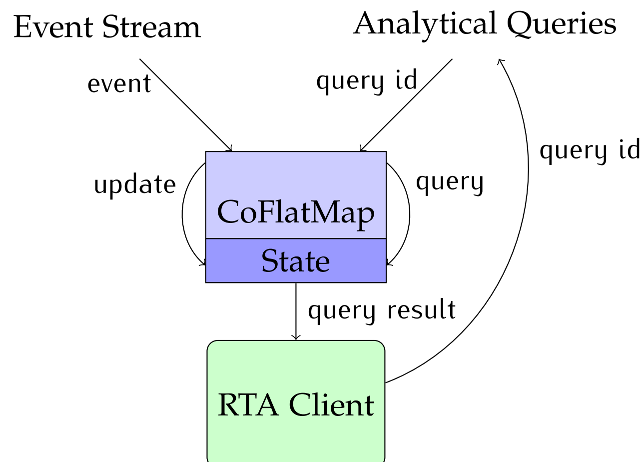


Figure 6: Hybrid processing in Flink. A CoFlatMap operator interleaves events with analytical queries.

Another challenge is to run the analytical queries on the state maintained by the event processing pipeline. Flink does not provide a globally accessible state that can be used in such cases. States are only maintained at an operator level and cannot be accessed from outside. We solve this problem by processing both the event stream and the analytical queries in the same CoFlatMap operator as shown in Figure 6. Both streams are processed interleaved using two individual FlatMap functions that both work on the same shared state. This works as both functions are part of the same operator. Our implementation interleaves the two different streams on a partition basis. Since Flink follows the embarrassingly parallel paradigm, it is not designed to synchronize access across partitions. As described in [30], the Huawei-AIM workload does not require such a global synchronization since events are only ordered on an entity basis.

A powerful feature of Flink is its partitioning. Flink automatically partitions elements of a stream by their key and assigns the partitions to a parallel instance of each operator. Each instance of our CoFlatMap operator only receives the events for its partition and thereby maintains a part of the total state. The analytical queries, however, should run on the whole state. Therefore, we broadcast the queries to each CoFlatMap operator instance and run them on the individual partitions. The resulting partial results are merged in a subsequent operator.

In our experiments, we use Kafka to send queries since it integrates well with Flink and ensures that no queries are lost. It would also be possible to ingest the queries using a TCP client or other more sophisticated handwritten clients.

Table 4: Options to mitigate the network bottleneck depending on whether we control the clients and whether they run on the same machine as the database.

	local	remote
control	(1) redesign client protocols	(2) redesign client protocols
no control	(3) override system calls	(4) employ user-space networking

2.4 EXTENSIONS TO MMDBS

In this section, we explore how MMDBs can be extended to increase their streaming capabilities.

First, events need to be ingested into the system with maximum speed. To improve the ingest performance of MMDBs, we discuss the use of user-space networking in Section 2.4.1. Second, to scale with the number of events, a MMDB needs to be able to process events in parallel. In Section 2.4.2, we study the parallel processing of single-row transactions in MMDBs, using HyPer as an example. And finally, if a centralized MMDB cannot handle the event stream or concurrent analytical queries become too expensive, we need to be able to scale the system to multiple machines. We therefore discuss a distributed MMDB architecture, again using HyPer as an example, in Section 2.4.3.

2.4.1 Mitigating the Network Bottleneck

The throughput of MMDBs is often limited by the overhead introduced by their networking components. The costs for (de)serialization, the networking stack, and hardware become a bottleneck. Even on a single machine, the overhead of Linux's default TCP networking stack limits the number of message round trips between two processes to about 50,000 per second.

One example for network bound workloads are analytical queries with large result sets where the time for transferring data between the database and the client dominates query execution time. This bottleneck can be mitigated by using more efficient client protocols [173]. Another example are update heavy workloads where the database system could process many more transactions per second than its networking components can deliver.

The problem of inefficient networking can be addressed on different levels. When we have full control over clients, we could rewrite the entire communication logic, in particular the protocols. If that is not the case, we can still optimize the server by using more efficient networking stacks. And, orthogonal to that, if the clients run on the same machine as the server, we can apply even deeper optimizations.

Table 4 shows approaches for these four cases. When we control the clients (1 and 2), we can modify them to use more efficient protocols. For instance, instead of sending events using PostgreSQL compliant update statements, we could send a raw stream of events. Streaming systems such as Flink have dedicated input consuming operators for each job that implement specialized deserialization schemes. However, in a MMDB we generally cannot control the clients and do not want to break the compatibility with SQL protocol standards. Assuming we do not have control over clients running on the same machine as the database server (3), we can override their system calls (e.g., `read()` and `write()`) to communicate using shared memory instead of using domain or TCP sockets [199]. This is achieved by dynamically linking a shared library using `LD_PRELOAD`. OpenOnload [186], a high-performance network stack by Solarflare, uses this technique to transparently accelerate socket-based applications. With shared memory communication, we avoid expensive context switches between kernel and user space. When the clients now run on remote machines (4), we can only modify the database server. To also avoid context switches here, we can employ user-space networking and retrieve incoming packets directly from within user space instead of retrieving them from within the kernel and delivering them to the respective applications.

mTCP is a high-performance TCP stack that implements common I/O system calls, such as `epoll()` and `read()` [89]. mTCP bypasses the kernel by utilizing user-space I/O frameworks, including Intel's Data Plane Development Kit (DPDK) [48]. DPDK uses huge pages to avoid TLB overheads, maps the device's memory into user space using the user-space I/O (UIO) subsystem, and uses polling instead of interrupts. A downside of DPDK is that it requires exclusive access to the network interface, which prevents non-DPDK applications from receiving or sending packets. In practice, however, we can use a second network interface for non-database traffic. Cloud providers, including Amazon Web Services (AWS) and Microsoft Azure, recently added next-generation DPDK-compatible network interfaces making user-space networking commonly available [20, 185].

Figure 7 shows the intended architecture. Clients do not need to be altered and can still communicate using the regular TCP stack, while the database server employs mTCP. We show the performance impact of mTCP in Section 2.5.9. Another approach is to increase message throughput is to use Remote Direct Memory Access (RDMA) over InfiniBand. In contrast to the mTCP approach, RDMA over InfiniBand does not involve the CPU and allows the network interfaces to write directly to main memory. However, the downside is that it requires expensive networking hardware on both the client and the server side.

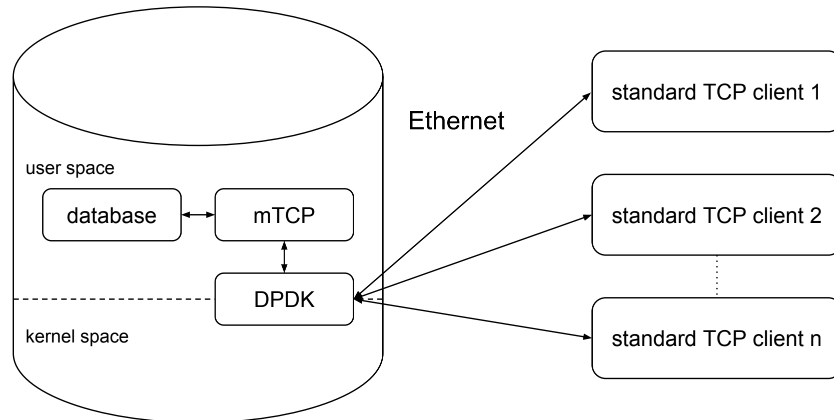


Figure 7: Using mTCP and DPDK on the server side to accelerate message throughput.

2.4.2 Improving OLTP Capabilities

In the past, MMDBs have either been dedicated to OLTP or OLAP [91, 25]. HTAP systems like SAP HANA [62] and HyPer [96] are positioned between these two extremes but prefer OLAP over OLTP. HyPer, for example, interleaves the execution of concurrent transactions but does not synchronize its data structures to avoid a performance penalty for analytical queries. Effectively, HyPer processes transactions using a single thread. HANA, on the other hand, stages updates in a delta store, which is periodically merged with the database. While these are valid strategies for read-mostly workloads, they are not optimal in a streaming setting with many concurrent OLTP events and OLAP queries that need to run on the current database state.

One approach that we have explored in [105] is HyPerParallel, an extension to the HyPer system that synchronizes concurrent transactions using optimistic locking [129]. HyPerParallel builds on HyPer's MVCC implementation [154]. To accelerate reads, this implementation organizes tuples in blocks (typically 1,024 tuples) and maintains a range of potentially versioned tuples per block. In addition to this range, HyPerParallel stores a version lock that encodes the block's version as well as its lock state. A reader reads the version before and after processing the block and repeats this process if the two versions differ (i.e., if the block has been altered by a concurrent writer). When the block is already locked (or has been locked in the meantime), the reader waits for its release before it (re-)starts scanning the block.

Note that this use of optimistic locking and MVCC is not specific to HyPer and can in fact be applied to any MMDB that uses MVCC. For more details on this technique, including synchronization of index structures and transaction management in MVCC, we refer the reader to [105]. Further, for state-of-the-art garbage collection in MVCC systems, we refer to recent work by Böttcher et al. [28].

Like HyPer, HyPerParallel uses a non-partitioned storage and processing model [123]. In other words, any processing thread can process a transaction which may alter any part of the data. This is in contrast to AIM and Flink, which partition both the data and the processing threads. HyPerParallel thus supports cross-partition transactions while AIM and Flink do not. There is, however, a challenge with HyPerParallel's processing model. Recall that in the Huawei-AIM workload an event corresponds to a single subscriber (a tuple in the database). As long as there is no skew in the accessed subscribers (i.e., accesses are uniformly distributed), there is no issue as concurrent threads likely access different subscribers. However, when introducing skew into the workload, write-write conflicts are more likely which may lead to aborts. We will show the performance degradation caused by these aborts in the evaluation (cf. Section 2.5.8). In addition, we will analyze HyPerParallel with a second execution model that partitions the workers according to the subscriber ID and thus effectively avoids any aborts due to serialization errors.

2.4.3 Scaling out Using Horizontal Partitioning

Most MMDBs are designed as standalone systems that do not support scaling out to multiple machines. The idea is to accept the inherent memory limitations of a single machine to avoid the introduction of any communication overhead. Once a system needs to communicate across machine boundaries, its performance is limited by networking hardware which usually has much higher latency and lower throughput than modern main memory. State-of-the-art networking technologies such as InfiniBand narrow this gap, however, they remain reserved to a small user group due to its significantly higher costs than commodity hardware (e.g., 1 Gbit Ethernet). We therefore do not further consider such next-gen hardware in this discussion and assume that cross-machine communication is expensive. Since distributed transactions[4] in such a setting are a costly operation (without sacrificing availability and/or consistency guarantees), we focus our discussion on the commonly used approach of horizontal partitioning with partition-local transactions. For a discussion of distributed transactions with next-gen networking technology, we refer the reader to related work by Zamanian et al. [220]. For better illustration, we discuss a reference implementation in HyPer, even though this approach is independent of HyPer's architecture and can be applied to every MMDB that has a server/client communication layer in place.

There have been multiple approaches to scaling HyPer to multiple machines. ScyPer [148] is a distributed version of HyPer that can scale with the number of analytical queries while sustaining a state-of-the-art transaction processing performance (>100,000 TX/s on TPC-C). ScyPer circumvents distributed trans-

4 Transactions that cross machine boundaries.

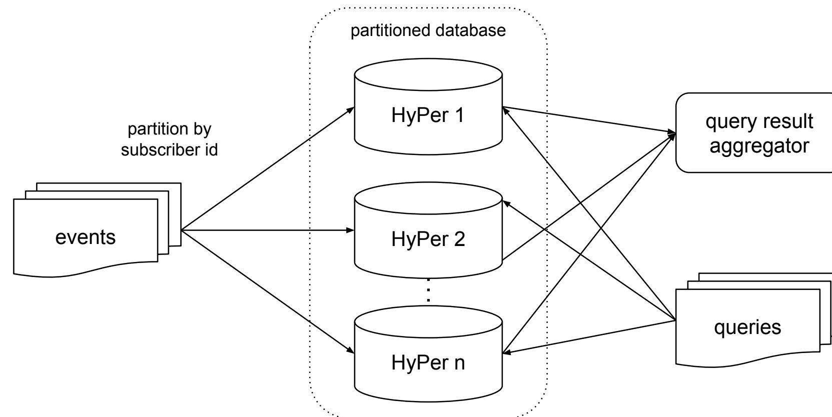


Figure 8: HyPerDistributed with multiple horizontal partitions (shards). Events are dispatched to the corresponding partition. Queries are sent to all partitions and aggregated on the client side.

actions by processing all transactions on a primary node and multicasting redo logs to secondary nodes. These secondary nodes are dedicated to analytical query processing. Roediger et al. introduce a distributed analytical query engine based on HyPer that replicates tables and achieves state-of-the-art results on the TPC-H benchmark [176]. While both systems achieve best-of-class results for their use cases, they do not consider partitioning the database and thus do not scale with the volume of the data or do not support transactions at all.

In this section, we introduce HyPerDistributed, a system based on HyPer that horizontally partitions the database similar to the distributed version of AIM (the hand-crafted C++ implementation of the Huawei-AIM workload).

Figure 8 shows HyPerDistributed's architecture. Due to the single-row updates of the Huawei-AIM workload, each event only affects a single partition. Queries, in contrast, are broadcasted to all partitions and are aggregated on the client side. We rewrote the queries to be able to execute them in a distributed setting (e.g., we replaced AVG with SUM and COUNT). Note that the queries are only aggregating data and thus their result sets are small. Hence, the client-side merging of the partial results does not cause a significant overhead. The communication between the client and the multiple HyPer nodes is implemented using PostgreSQL's C++ library (pqxx). Note that the pqxx protocol is verbose and thus inefficient when it comes to the serialization of query results. However, since the query results of this workload are small in size, we do not expect significant benefits from more efficient protocols. We refer the reader to [173] for an exhaustive study of the impact of communication protocols under analytical workloads with large query result sizes.

In Section 2.5.10, we study HyPerDistributed's performance on the Huawei-AIM workload and it compare against distributed versions of AIM and Flink. In future work, we could additionally employ redo log replay as suggested by

ScyPer to separate readers from writes. We also plan to centrally issue MVCC timestamps (for both transactions and queries) to allow for cross-partition consistency.

2.5 PERFORMANCE EVALUATION

We begin with a performance evaluation using the complete Huawei-AIM workload as described in Section 2.3. We drill down into the different aspects of the workload, including updates and real-time analytics. We then investigate the performance impact of the number of clients, the number of maintained aggregates, and skew. We finally study the impact of user-space networking and evaluate the systems in a distributed setting.

2.5.1 Configuration

We evaluate the different systems (cf. Table 5) on an Ubuntu 17.04 machine with an Intel Xeon E5-2660 v2 CPU (2.20 GHz, 3.00 GHz maximum turbo boost) and 256 GiB DDR3 RAM. The machine has two NUMA sockets with 10 physical cores (20 hyperthreads) each, resulting in a total of 20 physical cores (40 hyperthreads). The sockets communicate using a high-speed QPI interconnect (16 GB/s).

Table 5: Evaluated systems.

System	Version
HyPer	Jul 16, 2017
AIM	Same version as used in [30]
Flink	1.3.2

We place the clients on the same machine as the server and generate events and queries by one client thread each. Setting the total number of threads[5] was enough to run HyPer and Flink out-of-the-box. Conversely, AIM requires more tedious fine-tuning and server threads are allocated as explained in Section 2.3.2. As one can see from this allocation scheme, some workloads require more than one thread even in the most basic setting, which is why the measurements for AIM do not typically start at one thread.

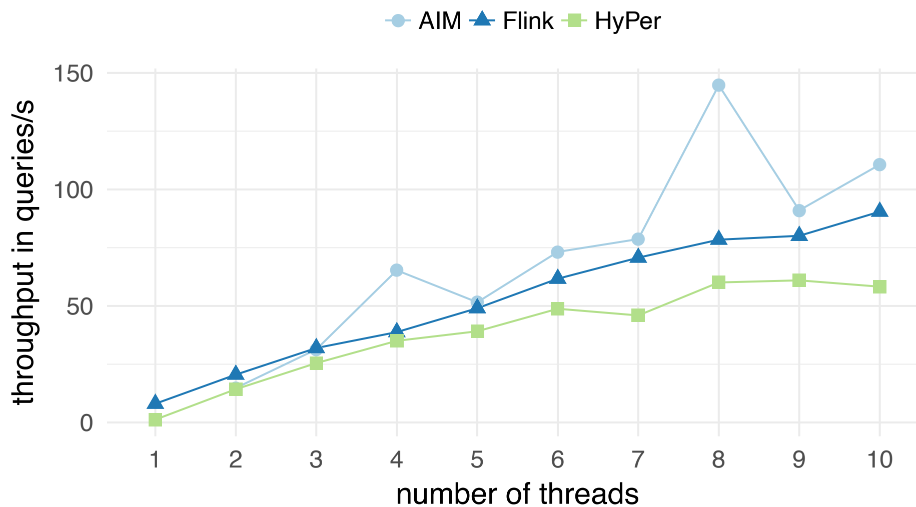


Figure 9: Analytical query throughput for 10 M subscribers at 10,000 events/s.

2.5.2 Overall Performance

Figure 9 illustrates the query throughput when running the full workload, which consists of 10 M subscribers, 10,000 events per second, and the seven analytical queries (cf. Table 3) where each of them is executed with equal probability. Further, daily and hourly windows are maintained leading to a total of 546 aggregates. AIM achieves the best performance. With two threads, it has a throughput of 14.8 queries/s and its best throughput, with eight threads, is 145 queries/s. The reason why AIM achieves its best performance at eight (and not at ten) threads is a NUMA effect: Since AIM statically pins threads to cores and allocates memory locally whenever possible, the total number of client and threads $(2 + 8 = 10)$ precisely fits on NUMA node 0. Hence, there is no communication to a remote memory region as it is the case for nine and ten threads. The spike at four threads probably relates to non-uniform communication paths between the cores on NUMA node 0. The spikes observed here are reproducable, and are, as we will see, also present in other workloads. Flink matches the performance of AIM for two threads and scales up to 90.5 queries/s using ten threads. HyPer achieves a throughput of 14.3 and 70.0 queries/s with two and nine threads, respectively. HyPer's throughput is lower than AIM's since it interleaves analytical queries with writes (i.e., writes block reads) while AIM processes them in parallel.

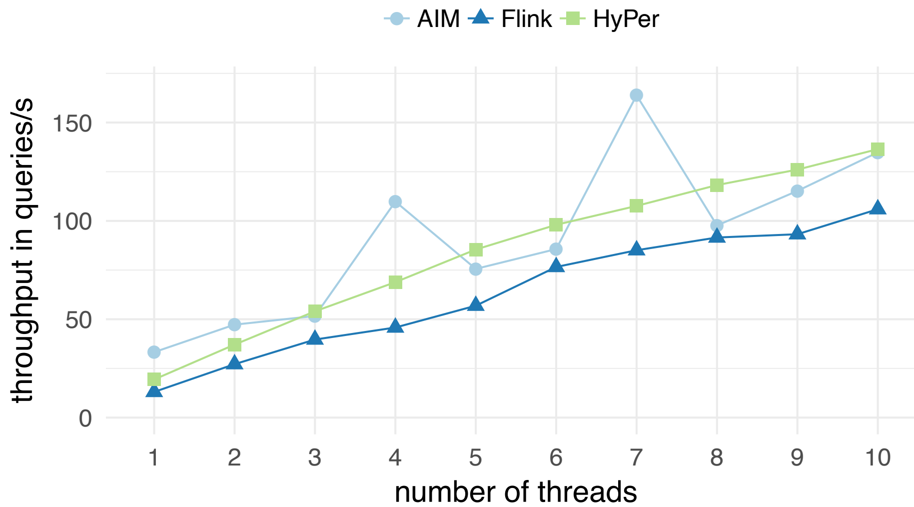


Figure 10: Analytical query throughput for 10 M subscribers.

2.5.3 Read Performance

Figure 10 shows the analytical query throughput for the different systems with an increasing number of threads without concurrent events. With one thread, HyPer processes 19.4 queries/s while AIM sustains a throughput of 33.3 queries/s. As we increase the number of threads, HyPer sometimes outperforms AIM and its throughput increases linearly while AIM shows the same spikes as before[6]. HyPer's maximum throughput is 136 queries/s with ten threads compared to 164 queries/s for AIM with seven threads. Flink's throughput is 13.1 queries/s using one thread and gradually increases to 106 queries/s with ten threads.

2.5.4 Write Performance

Figure 11 shows the event processing throughput of the different systems with an increasing number of event processing threads. This time, we evaluate the systems purely on the basis of their write throughput without running any analytical queries in parallel. Flink achieves the best write performance by far. Using one thread, it has a throughput of 14,400 events/s and the throughput scales to 222,000 events/s using ten threads. There are two reasons for this: (1) Flink partitions the state depending on the number of available processing threads. With this strategy, it scales well since there is no cross-partition synchronization involved. (2) Flink does not have any overhead introduced

5 Unless otherwise noted, we are always referring to the server-side threads.

6 AIM cannot be configured with zero ESP threads, which is why there is an additional idle ESP thread that we do not account for, but which nevertheless occupies its CPU. This is why the spike is at seven threads this time.

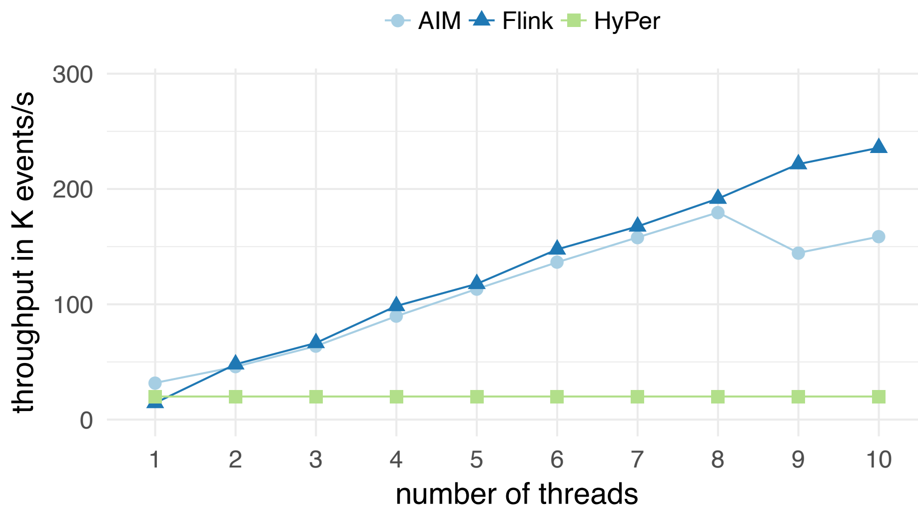


Figure 11: Event processing throughput with an increasing number of event processing threads.

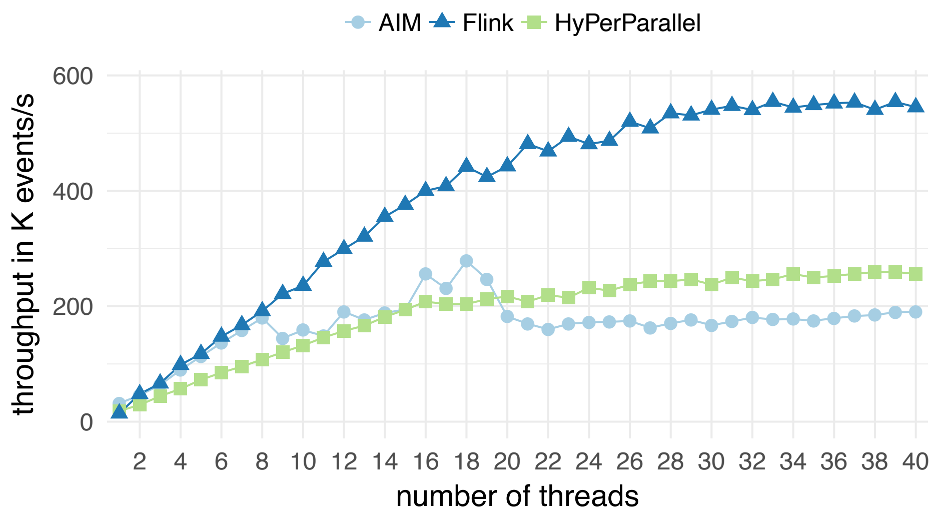


Figure 12: Event processing throughput with an increasing number of event processing threads (using optimistic locking in HyPerParallel).

by snapshotting mechanisms or durability guarantees. AIM processes 31,800 events/s using one thread and achieves a maximum throughput of 180,000 events/s using eight threads. Again, we see the NUMA effect described earlier. AIM also partitions the state to scale its write throughput, but since its *differential update* mechanism introduces an overhead, AIM does not perform as well as Flink. HyPer sustains a throughput of 20,000 events/s in all cases since it only uses a single thread to process transactions.

Next, we study the prototypical modifications to HyPer that we introduced in Section 2.4.2. These modifications allow HyPerParallel to parallelize trans-

actions by combining MVCC with optimistic locking. In this experiment, we vary the number of threads from one to 40 to also show the effect of oversubscription (using more threads than physical cores). Figure 12 shows that all three systems scale reasonable well with the number of physical cores. Flink's and HyPerParallel's write throughput even improves marginally with oversubscription while AIM's performance stabilizes at around 180,000 events/s.

The results show that HyPerParallel's optimistic locking approach achieves the same scalability as Flink and AIM which do not need to perform any synchronization due their to static partitioning. At the same time, HyPerParallel guarantees global consistency using MVCC while the other systems can only guarantee a consistent state per partition.

2.5.5 Query Response Times

In this experiment, we measure the response time for each of the seven analytical queries with and without concurrent writes (10,000 events/s) using four threads. Figure 13 shows the individual query response times. HyPer's performance degrades the most when writes are added to the query processing workload. The reason is that HyPer interleaves analytical queries with writes. As shown in the previous section, HyPer's write throughput is limited to 20,000 events/s and does not scale for multiple threads. Thus, an event throughput of 10,000 events/s blocks the query processing for about 500 ms every second. The query processing can only happen in the remaining 500 ms. AIM does not experience the same performance degradation since it performs writes and reads in parallel using the *differential updates* approach. Flink's performance does not drop much when adding 10,000 events/s as its (parallel) write throughput is so high that the analytical queries remain almost unaffected. However, we expect a higher performance degradation in Flink's analytical performance when increasing the number of events per second as Flink lacks efficient snapshotting mechanisms.

2.5.6 Impact of Number of Clients

Figure 14 shows the analytical query throughput with an increasing number of clients using ten server-side threads. HyPer performs the best of all systems and achieves a maximum throughput of 276 queries/s with ten client threads. HyPer's performance improves with multiple clients since it interleaves the execution of analytical queries (cf. Section 2.3.2). AIM's peak throughput is 218 queries/s with eight client threads. The gradual increase in the throughput shows the effect of the *shared scan* technique as AIM can now batch queries from multiple clients and process them all at once. The fact that the performance drops after eight threads shows that batching is only beneficial up to

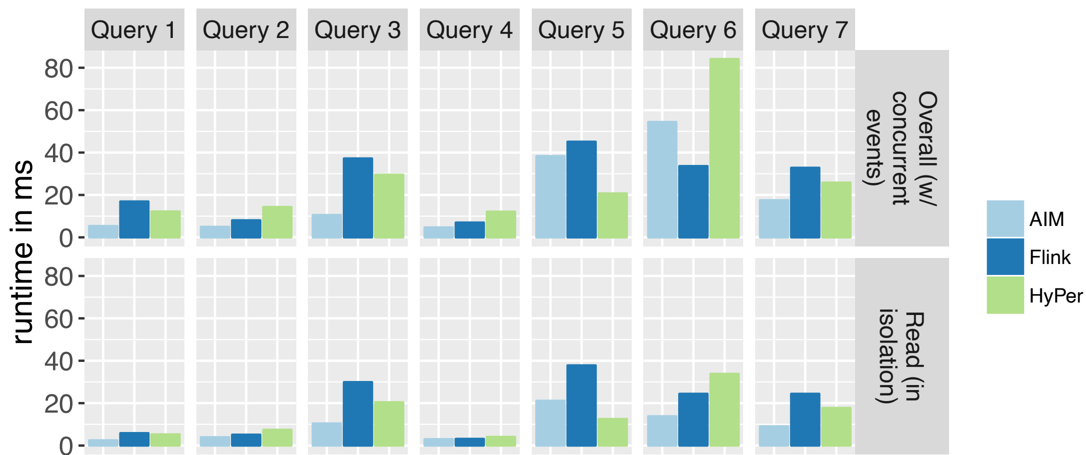


Figure 13: Query response times.

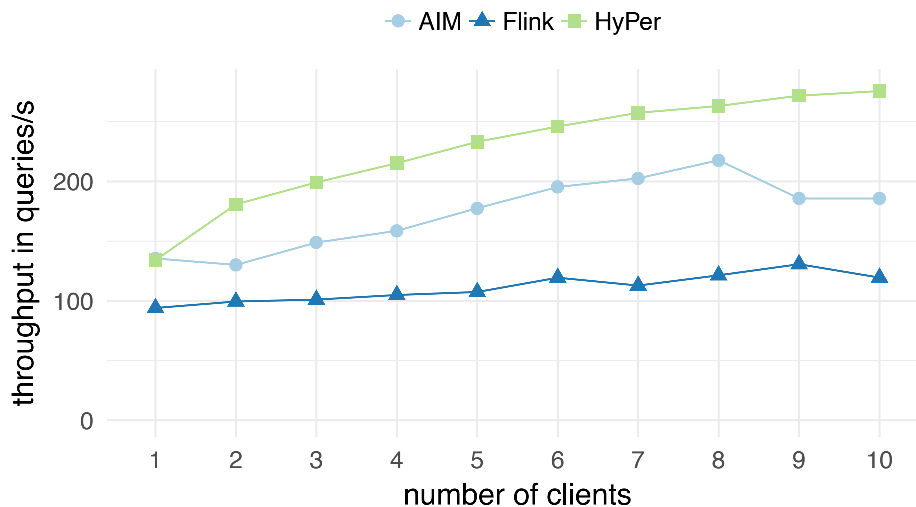


Figure 14: Analytical query throughput with an increasing number of clients.

a certain point. Flink executes analytical queries as follows: Once a worker completed its part of the query, i.e., processed the query on its partition of the state, the worker can continue with the next query. The worker does not have to wait until the other partitions have been processed and the partial query results have been merged. For this reason, the idle time of threads decreases for more clients and the query throughput increases to 131 queries/s.

2.5.7 Impact of Number of Aggregates

In this experiment, we studied the impact of the number of aggregates being maintained. We measure the overall as well as the write performance of AIM, HyPer, and Flink while maintaining 42 instead of the original 546 aggregates.

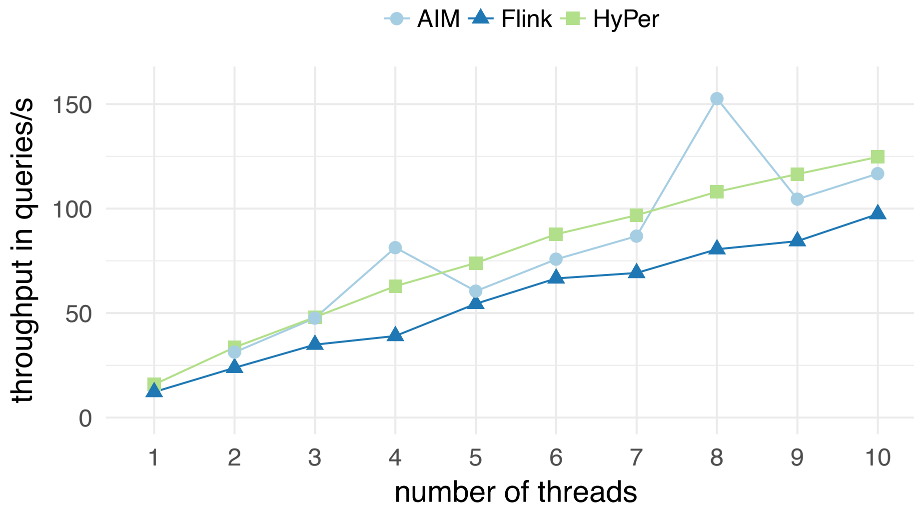


Figure 15: Analytical query throughput for 10 M subscribers and 42 aggregates at 10,000 events/s.

Figure 15 shows the analytical query throughput for 10 M subscribers and 42 aggregates at 10,000 events/s. Again, AIM achieves its best performance at eight threads (cf. Section 2.5.2) whereas Flink and HyPer do not experience such spikes. In contrast to the overall workload with 546 aggregates, HyPer achieves a higher performance than Flink throughout this experiment. The gain in HyPer's performance is expected since writes are now less expensive and thus singlethreaded phases are reduced. With ten threads, HyPer achieves a throughput of 125 queries/s (2.14× speedup over 546 aggregates) while Flink sustains 97.4 queries/s (1.08×).

Figure 16 shows the event processing throughput for 42 aggregates with an increasing number of event processing threads. Note that we have reduced the number of aggregates by a factor of 13. As expected, the throughputs improve significantly with less aggregates (cf. Section 2.5.4). With one thread, AIM and HyPer achieve a throughput of 227,000 (11.4×) and 228,000 events/s (9.62×), respectively, whereas Flink sustains 766,000 events/s (25.5×). With ten threads, AIM and Flink reach a throughput of 1,000,000 (7.69×) and 2,730,000 events/s (9.51×), respectively. HyPer's performance does not increase with more threads since it does not parallelize transactions.

2.5.8 Impact of Skew

In this experiment, we study the effect of skew on the event processing performance of the different systems. In the original Huawei-AIM workload, the subscriber IDs (keys) of events follow a uniform distribution. However, we presume that in reality entity IDs are skewed (i.e., there will be subscribers

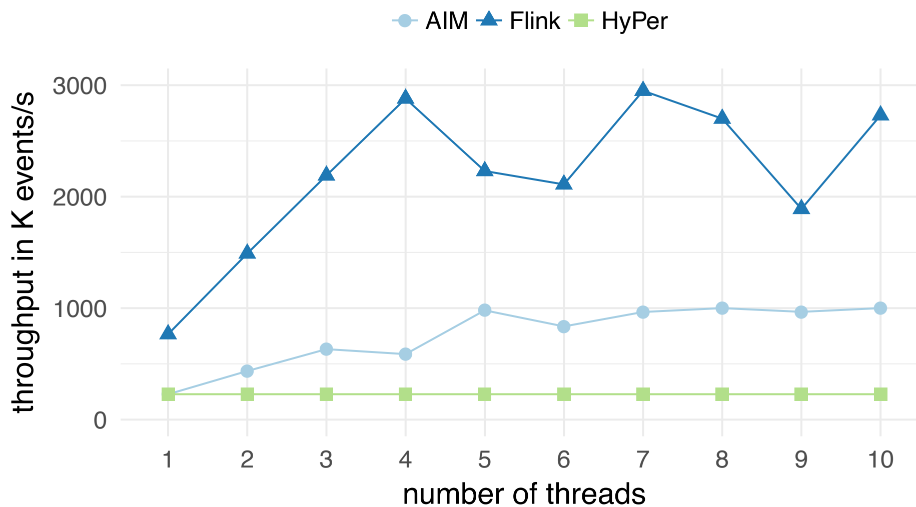


Figure 16: Event processing throughput for 42 aggregates with an increasing number of event processing threads.

that make significantly more calls than others). Therefore, we introduce a new flavor to the Huawei-AIM workload that generates events following a Zipfian distribution.

We analyze two different execution models in HyPerParallel, its default one that assigns events to processing threads in a round-robin fashion (HyPerP-RoundRobin), and a second one that partitions the workers according to the subscriber ID of the event (HyPerP-NoConflicts). In the latter model, there cannot be any write-write conflict between two threads that may lead to an abort. However, there can still be contention on indexes and locks (as subscribers may reside in the same MVCC block).

In Figure 17, we compare the two execution models of HyPerParallel to AIM and Flink. We normalize the chart using the throughput achieved without any skew.

Up to a Zipf factor of 0.75, the throughput of all systems stays rather stable. Starting from a Zipf factor of 1.0, AIM benefits most from the skew while Flink and HyPerP-NoConflicts are not affected until a Zipf factor of 1.25. HyPerP-RoundRobin, in contrast, starts to suffer from many aborts and its throughput is almost halved. At a Zipf factor of 1.25, AIM's and Flink's throughput starts to drop, while HyPerP-NoConflicts benefits from the increased skew.

When the amount of skew increases, two opposing effects can be noticed. Some subscribers are updated more frequently and thus their records are more likely to be cached. Thus, the performance of all systems slightly increases. Under high contention, the performance of the systems diverges. To understand the differences in the results, we have to revisit the design of the systems.

HyPerP-NoConflicts operates on a shared state (database) and only partitions the processing threads. In contrast, both AIM and Flink also partition

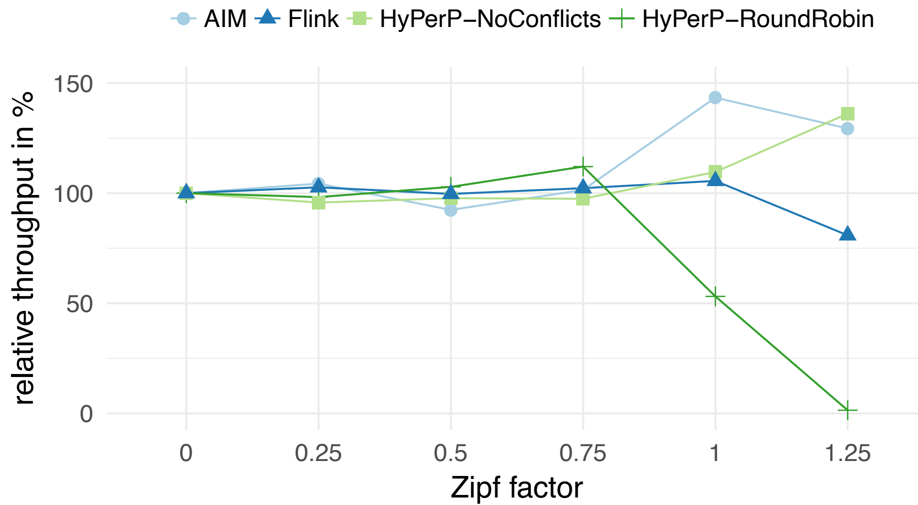


Figure 17: Relative event processing performance for 546 aggregates with increasing skew using 10 event processing threads (Zipf factor = 0 represents a uniform distribution).

the state. At a certain point, the amount of work per thread (i.e., per partition) becomes imbalanced. In the worst case, all updated subscribers reside in the same partition and thus all updates are executed serially by a single thread.

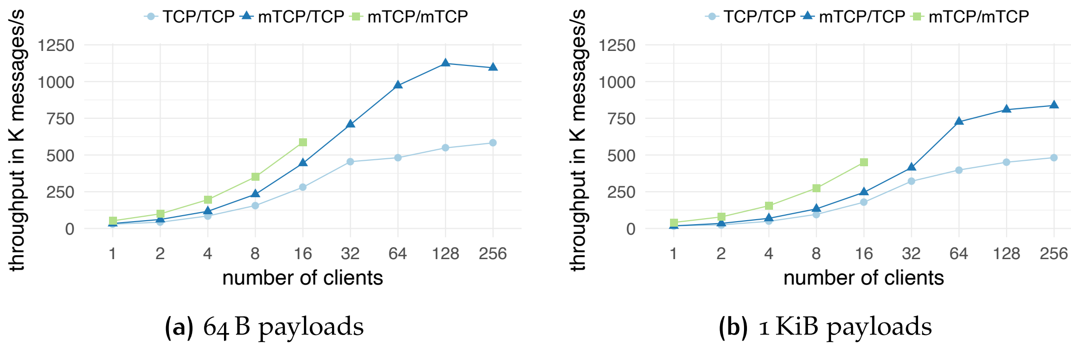
AIM's approach is slightly different from Flink's as it maintains an additional delta hash map per partition. Updates are first written to the delta before being periodically merged into the partition. With high skew, it is likely that the same subscribers are updated multiple times. Thus, when a customer is updated that already exists in the hash map, its entry can be reused and it is likely that the subscriber still resides in the cache. Further, the total size of the delta is smaller when fewer subscribers are updated, which reduces merge costs.

Flink does not maintain such a delta and instead updates data in place. The benefits of skew are not as high as for AIM and are already outweighed at a Zipf factor of 1.0 by the imbalance of work across partitions.

HyPerP-NoConflicts is similarly skew resistant as AIM. However, an advantage of HyPerP-NoConflicts is that it does not physically partition the database: the entire state is shared. This enables it to execute cross-partition transactions.

2.5.9 Impact of User-Space Networking

The goal of this experiment is to highlight the impact of user-space networking (cf. Section 2.4.1) and to motivate its adoption in MMDBs. We conduct the experiments on two identically configured Xeon machines that both run Ubuntu 16.04 and are equipped with a Intel Xeon E5-2680 v4 CPU (2.40 GHz, 3.30 GHz turbo) and 256 GiB DDR4 RAM. Both machines have a DPDK-comp-



(a) 64 B payloads

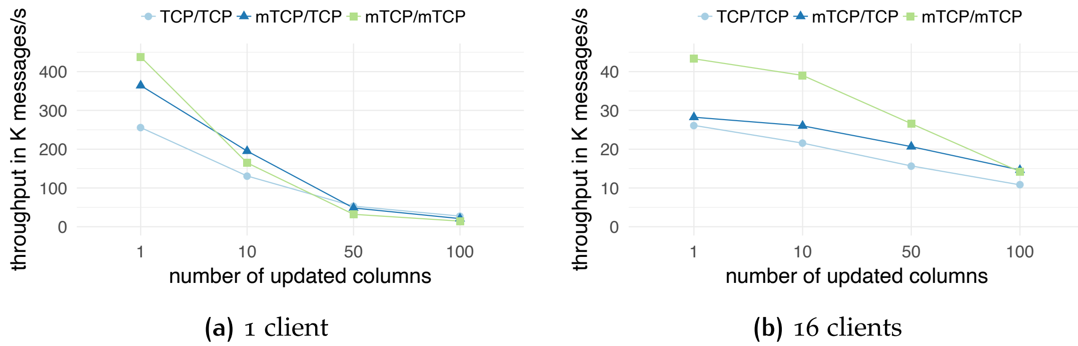
(b) 1 KiB payloads

Figure 18: Number of round trips in K messages/s for two different payload sizes with an increasing number of client threads (`mTCP/mTCP` = server and clients use mTCP, `mTCP/TCP` = server uses mTCP and clients TCP, and `TCP/TCP` = server and clients use TCP).

atible Intel X520 DP 10Gb DA/SFP+ network interface card (NIC) and are directly connected using a SFP+ cable. The average round trip latency reduces from 0.08 ms to 0.03 ms by using mTCP on the server side.

We have integrated mTCP (with its DPDK backend) into AIM's distributed version. However, as it turns out, AIM's event processing is actually not network bound. Meaning even a simple TCP communication (using Linux's kernel driver) can saturate AIM. The reason for this is the potentially large number of aggregates that is updated for each incoming event. We therefore have designed our own microbenchmark to show the impact of networking on a MMDB's performance. We have implemented standalone mTCP server and client programs that exchange 64 B and 1 KiB payloads. On the server side, a single thread listens for incoming messages. In a first experiment, the server immediately returns incoming messages back to the client without doing any further processing. On the client side, we increase the number of threads to eventually saturate the server. Thereby, we simulate a typical database scenario with multiple clients and one server.

Figure 18 shows the throughput for two different payload sizes. For both payload sizes, mTCP has a significant effect on message throughput. With 64 B payloads and 128 clients, `mTCP/TCP` achieves a peak throughput of 1.12 M messages/s (0.53 Gbit/s) while `TCP/TCP` can only sustain 0.58 M messages/s (0.28 Gbit/s). Note that we can achieve this 2.05× performance improvement by only modifying the server and leaving the clients untouched. When we also use mTCP on the client side, we can achieve even higher numbers. Note that mTCP does not support more than 16 cores due to NIC hardware queue restrictions. With 1 KiB payloads and 256 clients, there is a difference of 1.74× between `mTCP/TCP` and `TCP/TCP`. The lower impact of mTCP for larger payloads is expected since both the server and the clients have to do more work (e.g., copying more data from device to main memory). Note that the maximum



(a) 1 client

(b) 16 clients

Figure 19: Number of round trips in K messages/s for two different client thread counts with an increasing number of updated columns.

throughput of `mTCP/TCP` of 0.84 M messages/s (6.39 Gbit/s only considering payloads) is close to the physical limit of our 10 Gbit link.

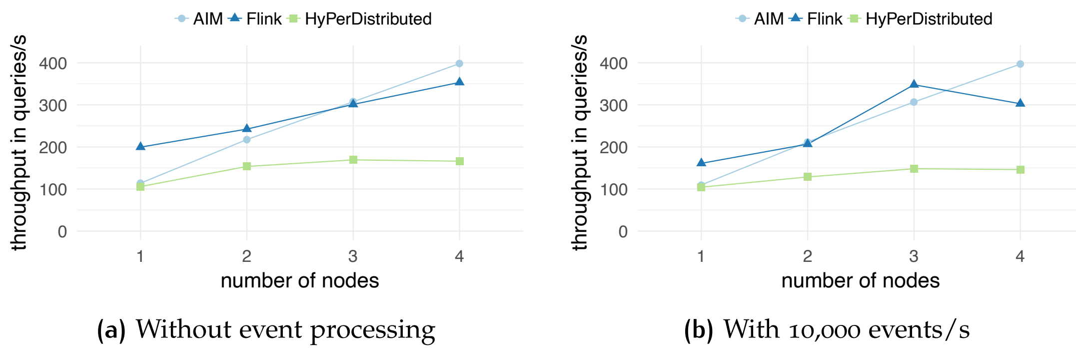
In a second experiment, we fix the payload size to 64 B and add lightweight server-side processing (as opposed to the heavyweight updates of the Huawei-AIM workload) to simulate a more realistic database scenario. For each incoming event, we perform an update operation in a column store with an increasing number of columns (1, 10, 50, and 100) with each column containing 1M rows. We uniformly choose one row to be updated.

Figure 19 shows the throughput for 1 and 16 client threads. For both client thread counts, mTCP has a positive impact on message throughput for up to 50 updated columns. By adding more columns, the server-side processing becomes increasingly expensive and at some point it dominates the (network) throughput. Up to this point (in this case 10 columns), the results clearly show a positive impact of user-space networking. Our experiments suggest that MMDBs should employ user-space networking for workloads with fast transactions or with low-latency queries (e.g., point lookups).

2.5.10 Distributed Setting

In this experiment, we study how the systems scale to multiple nodes. We use four identically configured machines (cf. Section 2.5.1) and commodity 1 Gbit Ethernet hardware. To compare the centralized MMDB HyPer with AIM and Flink, we implemented a distributed version (HyPerDistributed) that can execute the Huawei-AIM workload (cf. Section 2.4.3). All systems partition the state horizontally. Events are partitioned by subscriber ID and are sent to the corresponding node. We configure the systems to use all available 40 hyperthreads on each node. The client is placed on one of the four nodes.

Figure 20 shows the analytical query throughput with and without event processing. Due to its specialized and efficient networking protocol, AIM almost scales linearly from 110 to 397 queries/s. Up to three nodes, Flink shows



(a) Without event processing

(b) With 10,000 events/s

Figure 20: Analytical query throughput for 10 M subscribers with an increasing number of nodes.

a similar scalability before its performance decreases. This indicates that the communication between the nodes becomes a bottleneck. Flink's networking performance might be improved by implementing more specialized serialization schemes. At this point, we use its default serialization functions that might not be optimal for complex data types. We want to point out that Flink's infrastructure eases the development and the deployment of applications. An application can be deployed to a cluster without making any code changes. This allows local development and eases debugging. When deploying an application, Flink automatically compiles, optimizes, and distributes it across the cluster.

HyPerDistributed does not scale as well as AIM and Flink. We have investigated this further in a standalone benchmark and found that HyPer's analytical query throughput does not proportionally increase with less data. In other words, halving the data size does not lead to twice the throughput. Thus, even with perfect networking, distributing the workload over x machines cannot increase the performance by a factor of x. We account this to HyPer's single-node parallelization overhead. Its morsel-driven parallelism seems to be inferior to the static partitioning applied by the other systems. Note that the client of the Huawei-AIM workload blocks when sending queries (i.e., the result of the previous query has to arrive before sending the next query). Due to this request-reply pattern, a processing delay of a single node may have a significant impact on a system's overall performance. Therefore it is crucial to ensure deterministic query runtimes (e.g., by running the system in a performance isolated environment). Further, network latencies have a high impact on the scalability of a system. In Section 2.4.1, we outlined how this network bottleneck can be addressed.

We do not show the write-only numbers here as writes do involve any post-processing and therefore are embarrassingly parallel. Thus, the challenge for optimal write performance is more a question of a fast, scalable client and efficient networking (cf. Sections 2.4.1 and 2.5.9). For instance, for Flink it

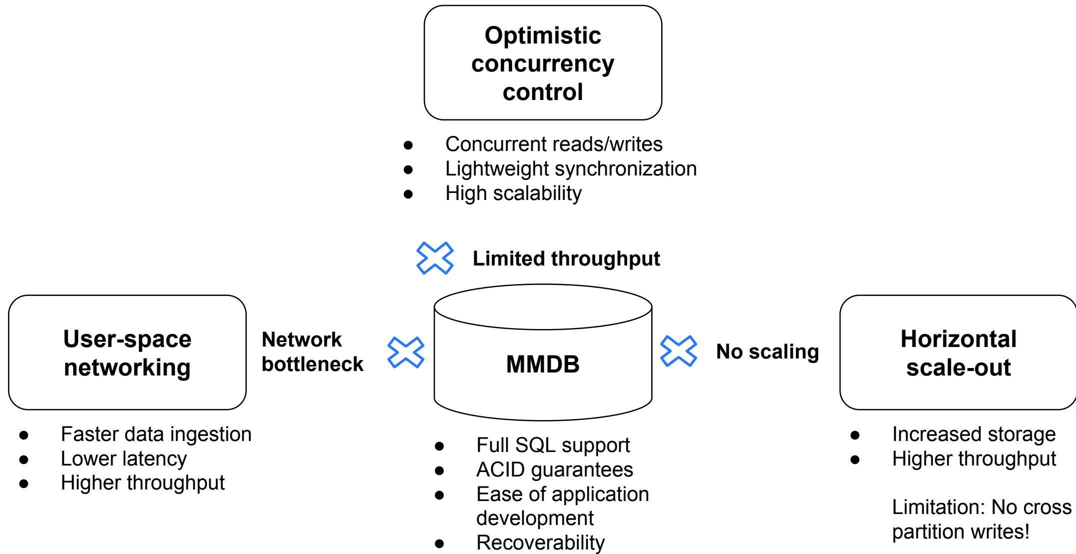


Figure 21: Overview of possible extensions to MMDBs and their benefits.

is important to add additional Kafka partitions for each parallel worker that should fetch events. If there is only one Kafka partition, only one thread can receive events.

2.6 CLOSING THE GAP

We have shown that general-purpose MMDBs perform fairly well on streaming workloads. Nevertheless, our experiments indicate that there is still a gap between the performance and usability of MMDBs and modern streaming systems, such as Flink. In this extended version, we have started to close this gap and have identified and evaluated extensions to MMDBs that increase their streaming capabilities (cf. Section 2.4).

As shown in Figure 21, we propose a threefold approach to improve the overall performance of MMDBs: (a) improve networking, (b) use all cores on a single machine, and (c) distribute load across multiple machines. In the following, we will describe each of these aspects in more detail.

In Section 2.4.1 we have highlighted a few options to mitigate the network bottleneck in MMDBs depending on whether we have control over clients and whether communication is local or remote. One such option is to employ user-space networking such as mTCP to increase message throughput by avoiding expensive context switches between kernel and user space (numbers can be found in Section 2.5.9). With the support of DPDK-compatible network interfaces by cloud providers such as AWS and Microsoft Azure, this seems to be a promising direction going forward. Another option to increase networking

performance is to employ more efficient client protocols, which is an area of open research [173].

AIM, Flink, and Tell are capable of processing events in parallel, whereas HyPer processes transactions in a single thread. To match their scalability, HyPer would need to be extended with parallel single-row transactions, which are less complicated to parallelize than full transactions. We have experimentally implemented a version of HyPer that can parallelize transactions (cf. Section 2.4.2) and have demonstrated its competitiveness.

MMDBs also need to be able to distribute writes across multiple machines. HyPer, for instance, could employ a similar strategy as Flink, which partitions the event input stream and distributes it across nodes. Towards this end, we have implemented a distributed version of HyPer that can handle the Huawei-AIM workload (cf. Section 2.4.3). This architecture could be further extended by employing ideas from the ScyPer architecture [148], where transactions are processed by the primary ScyPer node, which multicasts redo logs to secondary nodes.

From a usability perspective, modern streaming systems offer many features that help users to set up streaming applications, such as their out-of-the-box support for sliding and tumbling windows. On the one hand, MMDBs support arbitrary SQL allowing users to customize the analytical parts of their workloads and to issue ad-hoc queries. On the other hand, adding windowed aggregation functions using stored procedures is a cumbersome task. PipelineDB [168], which is built on top of PostgreSQL, solves this usability issue by extending SQL with streaming features but still cannot match the performance of dedicated streaming systems as we found in early experiments. In addition to out-of-the-box streaming features, modern streaming systems allow users to add custom code. There has been work to allow for the same in MMDBs, such as the integration of high-level programming languages (using user-defined functions). These additions, however, still do not allow for the same flexibility as writing plain old Java code. These limitations are mainly caused by the multi-tenant nature of database systems and the security level that these systems need to fulfill. MMDBs would need to allow for optionally disabling security arrangements (e.g., enforcing access rights) in favor of better extensibility. Another mitigation path that MMDBs could follow is to simply add more streaming features to its SQL processing logic, namely, window-based semantics as proposed by PipelineDB and StreamSQL [188]. This is also a topic we plan to address in future research.

Besides extending MMDBs to better support streaming use cases, the gap between MMDBs and streaming systems could be closed from the other direction, which would mean extending streaming systems with additional storage management features and query mechanisms. There is ongoing work to make use of Apache Calcite [8] (a SQL parser and optimizer framework) to extend Flink with streaming SQL and query optimization capabilities. Cache and register lo-

cality are crucial for high query performance and they can both be addressed very efficiently by compiling query plans into native code [151]. While this was possible to achieve in systems like HyPer or Tell, which are written in C++ and LLVM, it is more difficult to implement using JVM-based systems such as the streaming systems evaluated in this work. Moreover, implementing efficient storage management capabilities is a tedious task in JVM-based languages because to ensure data locality, custom memory management has to be implemented outside the JVM heap.

2.7 CONCLUSIONS

In this work, we have evaluated a broad set of architectures to address *analytics on fast data*. We have performed an experimental evaluation including at least one representative of each architecture. Our experiences as well as the performance results indicate that there still exists a gap between MMDBs and dedicated streaming systems. We have started to close this gap and have identified and evaluated extensions to MMDBs. These extensions include networking optimizations, parallel transaction processing, skew handling, and a distributed architecture. We believe that these extensions can enable MMDBs to address a broad set of workloads.

3 GEOSPATIAL ANALYTICS

Excerpts of this chapter have been published in [102, 101].

3.1 INTRODUCTION

Connected mobility companies need to process vast amounts of location data in near real-time to run their businesses. For example, Uber needs to map locations of cars and passenger requests (points) to predefined zones (polygonal regions) for allocation and dynamic pricing purposes [203]. These polygonal regions are typically largely disjoint (non-overlapping) and mostly static. Points, on the other hand, are often not known a priori. Thus, the problem is how to efficiently find the polygons that contain an incoming point.

Traditionally, such point-polygon joins [87] follow the *filter and refine* approach. In this two-phase evaluation strategy, the filtering phase typically uses an index (e.g., an R-tree) on the minimum bounding rectangles (MBRs) of polygons and probes the index for each point to obtain a list of candidate join pairs. Then, in the refinement phase, expensive point-in-polygon (PIP) tests are performed to discard false matches.

We argue that the time has come to rethink this strategy: First, main memory is not a scarce resource anymore and modern machines offer multiple terabytes of memory. Combined with the city-centric model of geospatial applications (e.g., Uber), we show that it is possible to maintain highly fine-grained indexes for entire cities (e.g., Uber's operating zones) in main memory, dramatically reducing the number of CPU-intensive PIP tests. Second, geospatial positions, nowadays typically obtained by smartphones or wearables, are inherently imprecise [51]. Thus, we argue that it is in many cases admissible to trade off accuracy for performance. Based on these two insights, we transform the traditionally CPU-intensive problem of point-polygon joins into one that is bound by memory access latencies.

In contrast to the classical filter and refine approach, *true hit filtering* [31] identifies actual join pairs already in the filtering phase, and thus partially avoids expensive refinements. This is achieved by using additional approximations (such as inner rectangles [93]) to approximate the interior of polygons, so that when a point falls into an interior approximation, it can be safely deducted that the point is contained in the polygon.

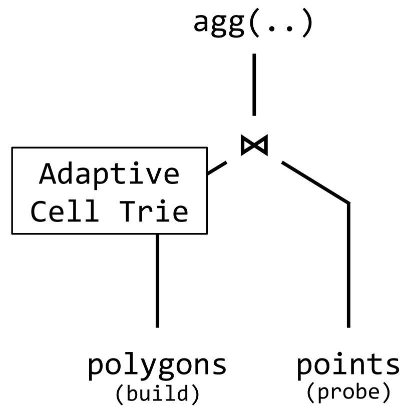


Figure 22: Polygons are indexed in a new trie data structure that is probed with dynamic points. The join result can be aggregated or materialized.

Building on this seminal idea, we present an improved algorithm that combines true hit filtering with quadtrees [108] to *holistically* index an entire set of polygons. This is in contrast to existing implementations of true hit filtering that approximate polygons individually [94, 61] or use non-hierarchical (single-resolution) grids [225, 16, 215]. In our approach, polygons are translated into a *single* set of multi-resolution grid cells that approximates their boundary and interior areas. To support efficient queries, we store one-dimensional identifiers of the cells in a new in-memory radix tree (trie) named *Adaptive Cell Trie* (ACT). We show that ACT is more query-efficient than previous approaches for indexing cell identifiers (e.g., B-trees, like in [94]).

Figure 22 shows a high-level query plan. The left input (polygons) is read first to create an ACT index, before the right input (points) is probed against that data structure to identify matches. The join result is either aggregated (e.g., by counting the number of points per polygon) or materialized for further processing. We focus on the case of static polygons and thus can pre-build the index.

Another distinguishing feature of our approach is that it can entirely avoid the expensive refinement phase by refining cells in the boundary areas until a user-defined precision is guaranteed. Naturally, this comes at the cost of higher memory consumption than traditional filter and refine approaches. However, as stated above, we argue that we can nowadays actually afford this higher memory consumption in exchange for higher performance.

Our approach can also provide accurate results by performing expensive PIP tests for points that are potential hits. To reduce their number, we adapt (train) our index based on historical data points to provide higher precision where it is actually needed. As we show in our experiments, our accurate algorithm performs very few PIP tests. Compared to a filter based on the polygons' MBRs, our index (trained with 1 M historical points) reduces the number of required PIP tests by >97% for a join between NYC taxi pick-up locations and

neighborhood polygons. This algorithm can also be used when ACT cannot guarantee the desired precision given a certain memory budget.

In summary, our contributions include:

- An algorithm that computes quadtree-based grid approximations for sets of polygons with precision guarantees
- A radix tree data structure (ACT) that is optimized for indexing cell identifiers: for a join of NYC's yellow taxi data with NYC's neighborhoods, we achieve a throughput of >50 M points/s per CPU core under a <4 m precision bound
- An evaluation of ACT in contrast to more traditional data structures, such as B-trees
- An accurate algorithm that trains the index structure based on historical data points
- An experimental comparison against state-of-the-art GPU-based point-polygon joins

The remainder of this chapter is structured as follows: Section 3.2 gives some background about the building blocks of our approach. Section 3.3 describes our approach and Section 3.4 presents the evaluation with real-world and synthetic data. Finally, we summarize related work in Section 3.5 before concluding in Section 3.6.

3.2 BACKGROUND

3.2.1 Location Discretization

Our approach relies on a quadtree-based (hierarchical) decomposition of space (the surface of the Earth in this case). This decomposition is static and thus *data independent*. We enumerate the quadtree cells using a space-filling curve (e.g., the Hilbert or the Z curve) to index them in a one-dimensional data structure. Our approach does not depend on a concrete space-filling curve. For our indexing strategy to work, the cell enumeration must only fulfill the property that child cells share a common prefix with their parent cell.

Figure 23 shows the hierarchical decomposition of two cells at levels i and $i+1$ and the corresponding bitwise representations that encode the cells' positions along the Hilbert curve. Each cell consists of four sub cells, which it completely covers. Child cells share a common prefix with their parent cell, allowing us to compute *contains* relationships using efficient bitwise operations.

Figure 23: Quadtree-based cell decomposition and Hilbert curve-based enumeration.

In our implementation, we use the Google S2 library [178] for mapping latitude/longitude coordinates to 64-bit cell identifiers, which we call *cell ids* in the following. A cell id encodes up to 30 levels with two bits per level.

3.2.2 Polygon Approximations

To obtain fine-grained polygonal approximations, we need a method that maps polygons to sets of quadtree cells (possibly at different levels). In particular, our algorithms take as input approximations of the boundary and interior areas of *single* polygons.

In our implementation, we use the S2 library to obtain these individual polygon approximations.

Figure 24 illustrates a covering (in blue) and an interior covering (in green) of a polygon. A point contained in a covering cell is either within or outside of the polygon while points that match interior cells are known to be within the polygon (true hits). The cell marked with (1) is one of the largest covering cells and only minimally intersects the polygon. Any point contained in this cell has at most a distance of $\sqrt{2} * \epsilon$ (with ϵ being the side length of the cell) to the polygon.

To allow for an efficient search, S2 stores the cell ids of a covering in a sorted vector. Besides sorting the cell id vector, it allows for the covering to be *normalized*. A normalized covering contains neither conflicting nor duplicate cells. Two cells are conflicting when one cell contains the other. Only when the covering is normalized can cell containment checks be efficiently implemented using a binary search on the sorted vector ($O(\log n)$). While binary search on a sorted vector is a good strategy for querying small collections of cells (e.g., the covering cells of a single polygon), it is not the most efficient way to search larger collections (e.g., coverings of multiple polygons). In this work, we store large cell collections in ACT, a query-efficient radix tree, and evaluate its performance compared to alternative physical representations (including a sorted vector and a B-tree).

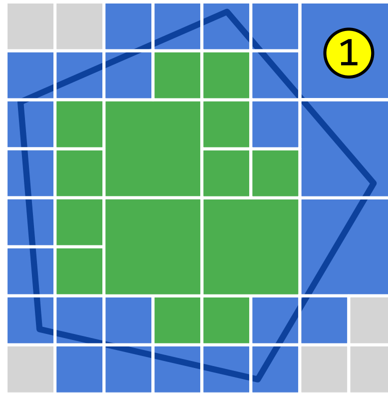


Figure 24: A covering (blue cells) and an interior covering (green cells) of a polygon.

3.2.3 PIP Test

A point-in-polygon (PIP) test determines whether a point lies within a polygon. Typically such a test is performed using complex geometric operations, such as the *ray-tracing algorithm* [77], which involves drawing a line from the query point to a point that is known to be outside of the polygon and counting the number of edges that the line crosses. If the line crosses an odd number of edges, the query point lies within the polygon. The runtime complexity of this algorithm is $O(n)$, n being the number of edges. While there are many conceptual optimizations to the PIP test, this operation remains computationally expensive since it processes real numbers (e.g., latitude/longitude coordinates) and thus involves floating point arithmetics.

3.3 APPROACH

In this work, we target the problem of mapping points to static, largely disjoint polygons. We show how to accelerate such joins by computing fine-grained cell-based approximations of sets of polygons and maintaining them in a query-efficient in-memory radix tree, which enables efficient cell lookups and significantly reduces (or even eliminates) expensive geometric tests.

In contrast to techniques that first reduce the number of candidate polygons using an index, e.g., an R-tree on the polygons' MBRs, and then refine candidates using geometric operations, our approach leverages true hit filtering [31] and identifies most or even all join pairs in the filter phase. On a high level, our approach first computes cell-based approximations of all polygons, called coverings and interior coverings, and merges them to form a *super covering*. Then, it stores these approximations in a specialized in-memory radix tree (named ACT) which allows for efficient lookups. Finally, ACT is probed for every point to obtain a list of *true* and *candidate* point-polygon pairs. The candidate pairs

are either refined by performing geometric computations to obtain an accurate result, or deemed to be part of the join result when small approximation errors can be tolerated.

The following provides more information about our indexing technique and the two geospatial join algorithms that are based on it: the approximate one that completely avoids expensive PIP tests while still guaranteeing a user-defined precision, and the exact one that reduces expensive computations by adapting to the expected point distribution. These algorithms allow us to trade memory consumption with precision (approximate approach) and performance (exact approach). Thus, they both favor modern hardware with large main memory capacities and high memory bandwidths. In summary, the key contribution of our indexing strategy is the novel combination of the super covering that approximates the polygons precisely, and the radix tree data structure that allows these approximations to be queried efficiently. With this design, we revisit the concept of true hit filtering in the context of modern hardware.

3.3.1 Adaptive Cell Trie (ACT) Indexing

Super Covering Computation

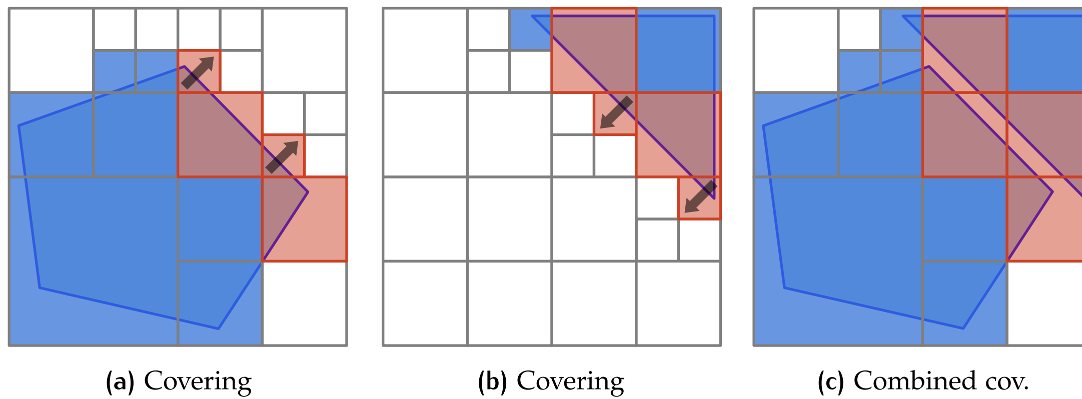
The super covering consists of a set of multi-resolution grid cells. All grid cells are disjoint in the sense that each geographical point is covered by at most one cell, even if two (or more) polygons overlap. A single cell of the super covering can therefore be associated with multiple polygons. The super covering maintains a list of *polygon references* for each individual cell. A polygon reference has two attributes:

POLYGON ID The identifier of the polygon that this cell references.

INTERIOR FLAG Whether the cell is an interior or a boundary cell of the polygon.

The precision of the super covering determines the selectivity of the index. When combining the approximations of the individual polygons, we need to take special care of conflicting cells[1] to not lose precision. However, this is challenging for two reasons. First, conflicts may occur between the cells of a covering of a given polygon and the cells of its interior covering. The interior cells always overlap some (if not all) covering cells. Second, when different polygons overlap or are close to each other, conflicts may occur between the cells of their coverings.

1 Recall that a conflict between two cells exists if one cell contains the other. We do not consider duplicate cells as conflicting.



(a) Covering (b) Covering (c) Combined cov.

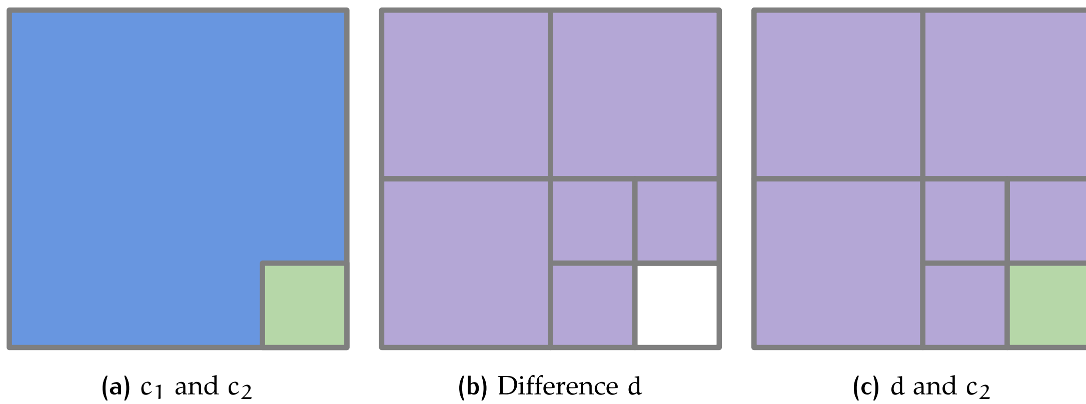
Figure 25: A combined covering may be less selective than two individual coverings. The arrows indicate that the cells will be expanded.

One approach for retaining the precision would be to not resolve such conflicts at all and maintain all conflicting cells. However, this would have the consequence that a query point could match with more than one cell, which would affect lookup performance. In a radix tree, this would mean that we would need to keep searching lower levels once we found a match at a higher level.

Ensuring that cells are non-overlapping results not only in higher lookup performance, but also in a more compact radix tree. The reason is that for a given entry in a tree node, we only need to differentiate between a pointer (to a child node) and a value. With overlapping cells, we would have to store a pointer *and* a value.

There are two obvious solutions for resolving a conflict between two cells c_1 and c_2, where c_1 is an ancestor of c_2 in the quadtree (c_1 contains c_2). One is to replace c_2 with c_1, which leads to a precision loss as shown in Figure 25. Figures 25a and 25b show the coverings of two individual polygons. The red cells have conflicts with cells of the other covering. Figure 25c shows a combined covering, where the originally smaller cells are subsumed by larger cells, causing a precision loss. The other solution is to replace c_1 with a set of smaller cells at the same level (i.e., of the same size) as c_2. While this retains the precision, it can significantly increase the number of cells in the combined covering.

Without compromising on precision, we would like to reduce the number of cells introduced. To solve this problem, instead of storing both conflicting cells c_1 and c_2, we compute their difference d and store c_2 and d. This has the advantage that there will not be any overlap between the indexed cells, and thus an index lookup will return *at most* a single cell. The side effect is that the total number of cells will increase since d consists of at least three cells. Figure 26 illustrates this precision preserving conflict resolution. Assume that c_1 (blue) and c_2 (green) are cells of two different coverings and that c_1 contains c_2. First, we compute d, which consists of six cells. We then copy all polygon



(a) c_1 and c_2 (b) Difference d (c) d and c_2

Figure 26: Precision preserving conflict resolution. c_1 is marked in blue, c_2 in green, and the cells in d in purple. Note that c_1 contains c_2.

references of c_1 to d and c_2 and omit c_1. Note that the cell count is increased by five. Overall, our approach retains the precision and the type (boundary or interior) of the individual cells as well as the mappings of cells to polygons.

Listing 1 outlines this algorithm. We iterate over all input cells and try to insert them into the super covering. When a cell already exists, this means that it is also part of another covering that has already been processed. When two cells conflict, this means that either the current cell covers the other cell or vice versa. These two cases may happen when polygons overlap or are close to each other, or when we first insert the cells of the covering of a given polygon and then the cells of its interior covering. To address these cases, we apply the precision preserving conflict resolution strategy described above. As mentioned earlier, this strategy increases the total number of cells. However, a more precise index reduces the number of expensive PIP tests and thus increases overall performance.

Figure 27 shows a super covering of neighborhoods in NYC's Jamaica Bay. Boundary (former covering) and interior cells are again marked in blue and green, respectively. Most of the area shown is covered by either interior cells or by no cells at all. Only in the unlikely event that a query point hits a blue (boundary) cell, we may experience false positives (approximate approach) or we will need to enter the refinement phase (exact approach).

Data Structures

To store the super covering and enable efficient queries over it, we use two data structures: (i) a specialized radix tree (ACT) that indexes the cells of the super covering, and (ii) a lookup table that maintains the (variable-length) polygon references. Both data structures are designed for in-memory processing and are optimized for lookup performance.

ACT is a specialization of a textbook radix tree that indexes 64 bit cell ids. We call it adaptive for two reasons: (i) it indexes cells of adaptive sizes (to

```
input:
  a list of coverings coverings // one per polygon
  a list of interior coverings interiors // one per polygon
output:
  // a list of (cell, polygon references)
  the super covering superCovering
procedure:
  for (covering in coverings) {
    for (cell in covering) {
      if (superCovering already contains cell) {
        add references of cell to existing cell
        continue
      }
      if (cell conflicts with existing cell in superCovering) {
        // cell is covered by existing cell or vice versa
        // resolve conflict
        c1 = ascendant cell // may be cell or existing cell
        c2 = descendant cell // may be cell or existing cell
        d = difference of c1 and c2
        add references of c1 to d and c2
        remove c1 from superCovering // only required if the existing cell is the ascendant
      cell
        add c2 and d to superCovering
        continue
      }
      add {cell, {covering.polygonId, interior flag=false}} to superCovering
    }
  }
  // ... same code for interior coverings (with interior flag=true)
```

Listing 1: Build precision preserving super covering.

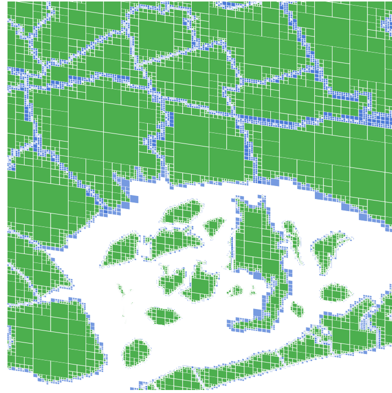


Figure 27: A super covering of neighborhoods in NYC's Jamaica Bay.

guarantee user-defined precision), and (ii) it can adapt to the expected point distribution. All adaptation is performed at build time. Once ACT is built, it is a static (immutable) data structure. We leave updates such as adding new polygons to an existing ACT for future work. However, we would like to point out that supporting updates is straightforward: In the build phase, cells of individual polygons are inserted one-by-one into ACT. The same procedure could be used to add new polygons at runtime, with appropriate synchronization between readers and writers. Code for removing polygons would follow the same logic, with the only difference being that we may want to (periodically) reorganize (i.e., compact) the lookup table.

We refer to the cell ids stored in the radix tree as *keys*. Each key denotes the path of a cell in the hierarchical grid. In the following, we first outline why a radix tree, in general, is a good choice for indexing quadtree cells, and we then explain how ACT differs from a general-purpose radix tree.

The main reasons why we choose a radix tree to index a super covering are (i) space efficiency and (ii) support for efficient prefix lookups. Compared to storing cell ids in a list, a radix tree avoids redundantly storing common prefixes, which reduces memory consumption. Prefix lookups, on the other hand, are required to find matching cells: The query point, which is a cell id at the most fine-grained grid level, is used to search for cell ids within the radix tree that share a common prefix (i.e., cover the query point). The runtime complexity of these lookups is in $O(k)$ with k being the key length, as opposed to the $O(\log n)$ of binary search that could be used on a sorted list. In other words, the number of node accesses in a radix tree is bounded by the maximum key length k_{max}, which is 60 when 30 quadtree levels are used (which is the case in our implementation). In practice, a lower k_{max} is often sufficient. For example, $k_{max} = 44$ allows for indexing cells up to level 22, which corresponds to a precision of less than 4 m (i.e., the distance between a point and a polygon in a false match is at most 4 m). A further advantage of the radix tree is that most queries can be answered using the upper levels of the tree: larger cells

use fewer bits and are thus indexed closer to the root node. In the likely event that a query point hits a larger cell, we can complete the tree probe sooner.

We now discuss the design choices of ACT. The fanout f of the radix tree controls space consumption and lookup performance. A fanout of four means that we consume two bits at every tree level. With that configuration, our data structure matches the quadtree scheme (each node has four children, cf. Figure 28 for an example). While this would ease the implementation, it would require up to 30 nodes to be accessed per lookup. With a higher fanout, we can reduce this number. To maximize lookup performance, ACT uses a *default* fanout of 256 (= 8 bits). Thus, each level in ACT corresponds to four levels in the quadtree (each quadtree level is encoded with two bits). Let g be the cell level granularity of ACT (with $f = 256$, $g = 4$). While a fanout of 256 may result in sparsely occupied trie nodes, it allows for efficient lookups as it reduces the height of the trie to k_{max}/g. With $f = 256$, the maximum number of node accesses is $\lceil 60/\log_2(256) \rceil = 8$ for 30 quadtree levels.

Now we exploit a property of the hierarchical cells that we index: We *extend* their cell ids (keys) such that the key length matches the granularity of ACT. This process involves replacing a cell that we want to index with all its descendant cells at the next supported granularity level, and replicating the payload of the original cell to the smaller cells. In other words, if a cell does not match the tree granularity, we recursively split it into smaller cells that cover the same area. The following holds for indexed keys (cells):

$$\mathit{level}(\mathit{cell}) \bmod g = 0$$

Each cell c for which this equation does not hold is decomposed into a set of smaller cells C, with $|C| = 4^{g-(\mathrm{level}(c) \bmod g)}$. This is possible since points are represented by cells at the most fine-grained grid level and use the maximum key length. Therefore, for a query point, it does not make a difference whether it matches with the originally inserted cell or with one of its descendant cells. This insight greatly simplifies the memory layout of a tree node and saves many CPU instructions: (i) we do not need to store the level with a cell, since all cells indexed in a tree node will have the same level, and (ii) a lookup in a node (an array) becomes a single offset access. Without this artificial key extension, we would need to perform multiple accesses per node to traverse all cell levels indexed in that node.

Figure 28 illustrates ACT indexing three polygons. While the example shows ACT with a fanout of four, by default we actually use a fanout of 256 to reduce the tree height. Every node thus consists of a fixed-sized array of 256 entries of 8 byte pointers. Entries that neither contain a child pointer nor a value point to a sentinel node indicating a false hit (no hit).

Values (i.e., polygon references) can be found in any level of the tree. This is because the indexed keys (64 bit cell ids in our case) typically use only a small fraction of the 64 bits with the remaining bits all set to zero. Larger cells

that use fewer bits are indexed higher up in the tree, possibly even in the root node. In our example, polygon a is indexed by a cell in the upper level, while polygons b and c are indexed by cells in the lower level. Instead of storing values in separate nodes (e.g., adjacent to the tree nodes), we use combined pointer/value slots like in [126]. This design consumes less space and avoids an unnecessary indirection. Here, we exploit another property of the cell ids that we index: Cells in the super covering are disjoint, therefore a tree lookup will return at most one result. Due to this property, we never need to store a pointer and a value in an array entry at the same time. Using *pointer tagging*, we differentiate between pointers and values. We therefore refer to both pointers and values as *tagged entries*.

As stated before, each cell is associated with a set of polygon references. Thus, each value stored in the tree has to identify such a set. The canonical design would be to make each cell point to an entry in a lookup table that stores the references. However, at least in the case of largely disjoint polygons, cells mostly reference only one or two polygons. Therefore, to eliminate additional indirections, when there are no more than two polygon references, we store these references directly in the tree. A tagged entry can thus be:

- An 8 byte pointer to a child or the sentinel node (recall that a pointer to the sentinel node indicates a false hit)
- An inlined polygon reference (a 31 bit value)
- Two inlined polygon references (two 31 bit values)
- An offset (a 31 bit value) into a lookup table indicating that there are at least three polygon references

We use the two least significant bits of the 8 byte pointer to differentiate between these four possibilities. For an inlined polygon reference, we differentiate between a true hit and a candidate hit using the least significant bit of the 31 bit value. Thus, we can effectively only store 30 bit polygon ids (i.e., can index up to 2^{30} polygons).

We have experimented with path compression, but have found that storing common prefixes with inner and leaf nodes only barely reduces the number of nodes. Thus, the additional cache miss to access the prefix does not pay off. We therefore only use a common prefix at the root level.

We have also considered introducing adaptive node sizes, as proposed by the adaptive radix tree (ART) [126]. However, experiments have shown that introducing a second (compressed) node type with four children (Node4 in ART) (i) saves only a negligible amount of space for our workload and (ii) has a significant performance impact (due to the additional instructions and branch misses for dispatching between node types [126]). Also, lookups in compressed node types are more expensive.

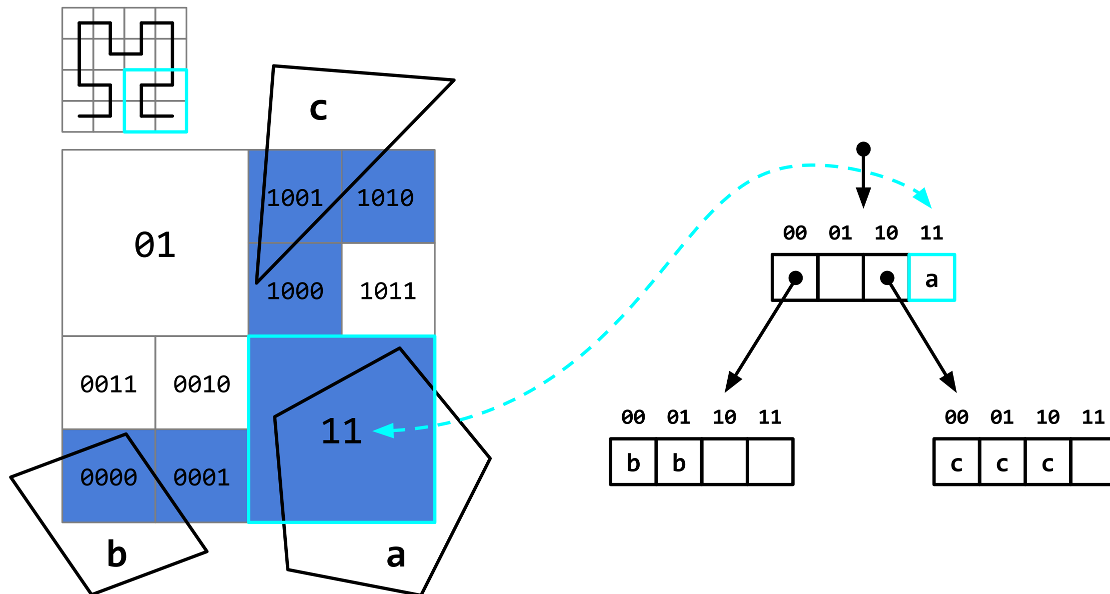


Figure 28: Adaptive Cell Trie indexing three polygons a, b, and c. Here, ACT uses two bits per level. In practice, we use up to eight bits (a fanout of 255) to reduce the tree height. Note that the figure only shows the cell rasterization for the part of the map that corresponds to the radix tree.

When a cell references more than two polygons, the tree contains an offset into a lookup table. Since cells often reference the same set of polygons, we only store *unique* polygon reference lists. The reference lists are split into two parts, a list with true hits and a list with candidate hits. Both lists contain polygon ids. The lookup table is encoded as a single 32 bit unsigned integer array. The offsets stored in the tree are simply offsets into that array. Each encoded entry contains the number of true hits followed by the true hits, the number of candidate hits, and the candidate hits.

Index Probing

An ACT lookup returns, at most, one cell mapping to a set of polygon references. Listing 2 shows the probe algorithm. While traversing the radix tree does not involve any key comparison, a comparison is performed to check whether the returned tagged entry contains a payload. For that, we need to differentiate between (i) one polygon reference, (ii) two polygon references, and (iii) an offset. In the first case, we check whether the polygon reference is invalid, which indicates a false hit. Otherwise, we extract the interior flag (the least significant bit of the 31 bit payload) and the polygon id and return the reference. In the second case, we extract and return both references. Only in the third case, we need to access the lookup table to retrieve the polygon references.

```
input:
  root node of ACT rootNode
  the cell id of the query point cellId
output:
  tagged entry taggedEntry
procedure:
  if (common prefix of rootNode does not match)
    return invalid entry
  level = 0
  currNode = rootNode
  bits = getBits(cellId, level++) // extract relevant bits
  // traverse the tree until we either hit the sentinel node or found a value
  while (taggedEntry = currNode.getEntry(bits) is a pointer) {
    if (taggedEntry points to the sentinel node)
      return false hit
    currNode = taggedEntry
    bits = getBits(cellId, level++)
  }
```

Listing 2: Probe Adaptive Cell Trie.

3.3.2 Approximate Join with Precision Bound

The complete point-polygon join algorithm is shown in Listing 3. It is essentially an index nested loop join, using our novel ACT index that makes the point-cell containment tests very efficient. For a given point, we retrieve the cell that contains it (if such a cell exists) and go over all references of this cell. When approximate results are sufficient, we omit the expensive refinement phase, simply treat all points contained in boundary cells as (approximate) hits, and immediately output the join pairs. In doing so, we introduce false positives. However, the distance of false positives from the polygon is bounded by the diagonal of the largest boundary cell: Any point contained in that cell has at most a distance of $\sqrt{2} * \epsilon$ (with ϵ being the side length of the cell) to the polygon. In order to control this distance, our approximate algorithm exposes a precision bound as a parameter to the user. Based on this bound, we compute the minimum cell level for boundary cells. For example, to guarantee a 4 m precision, the largest boundary cell can at most have a diagonal of 4 m, which corresponds to a minimum cell level of 22 in our implementation (i.e., cell level 21 would be too coarse-grained). We replace all boundary cells in the super covering with their descendant cells at the required level. For each of these descendant cells, we determine whether they intersect, are fully contained in, or do not intersect polygons at all, and update ACT accordingly: We remove the original cell c_0 from ACT and insert *only* those descendant cells that intersect or are fully contained in polygons. The new cells may reuse the

```
input:
  points points // lat/lng coordinates and cell ids
  polygons polygons // lat/lng coordinates of vertices
  root node rootNode
  lookup table lookupTable
output:
  list of join pairs pairs // point/polygon pairs
procedure:
  for (point in points) {
    taggedEntry = probeAdaptiveCellTrie(rootNode, point.cellId) // cf. Listing 2
    if (taggedEntry is invalid)
      continue
    references = getPolygonReferences(lookupTable, taggedEntry) //
     returns a list of polygon references
    for (reference in references) {
      polygonId = reference.polygonId
      polygon = polygons[polygonId]
      if (reference is true hit) {
        add {point, polygon} to pairs
      } else { // candidate hit
#ifdef __APPROX
        // treat candidate hit as true hit
        add {point, polygon} to pairs
#else
        // EXACT: enter refinement phase
        if (polygonCoversPoint(polygon, point)) // PIP test
          add {point, polygon} to pairs
#endif
  } } }
```

Listing 3: The join algorithm.

lookup table entry of c_0 or create their own in the event that they only map to a subset of c_0's polygons.

Note that [215] makes use of a similar distance-based precision bound, however, uses a single-resolution grid. When it is not possible to maintain a sufficiently fine-grained index within a certain memory budget, the user can fall back on our accurate approach, in which we train the index with historical data points.

3.3.3 Accurate Join

When applications require accurate results, or when we cannot build an index that satisfies a user-defined precision without exceeding a memory budget, we use an approach that may enter the expensive refinement phase (cf.

Listing 2). To minimize the number of (expensive) PIP tests, we increase the precision of the index by adapting it to the expected point distribution. Since we make use of true hit filtering, a finer-grained index allows us to identify more join partners during the filter phase.

Index Training

To minimize the likelihood of PIP tests, we train the index to adapt to the expected distribution of query points. We train ACT with historical data points (e.g., from a previous year) which has the effect that popular areas that expect more hits are approximated using a more fine-grained grid than less popular areas. This training process replaces *expensive* cells with up to four of their child cells. We define expensive cells as cells that map to polygon reference sets with at least one candidate hit. When we hit such a cell during the join, we need to perform expensive PIP tests.

Specifically, the training works as follows: When a training point hits an expensive cell, for each of its four child cells we check whether they intersect, are fully contained in, or do not intersect the referenced polygons at all, and update ACT accordingly. The cell replacement procedure is the same as for the approximate algorithm (i.e., remove original cell, insert descendant cells, and update lookup table, cf. Section 3.3.2) with the only difference being that we always replace an expensive cell with its direct children one level below. We do not replace a cell with even smaller cells to be more robust against outliers. In practice, we would stop refining the index once a user-defined memory budget is exhausted. In this work, we focus on training the index in a dedicated training phase. Training the index at runtime would introduce additional concurrency and buffer management issues that we leave for future work. We show the effect of training the index in Section 3.4.5.

3.3.4 Implementation Details

Join Predicate

Our current implementation follows the semantics of the `ST_Covers` join predicate (cf. PostGIS [170]). `ST_Covers` evaluates whether one geospatial object (e.g., a polygon) covers another (e.g., a point).

Individual Polygon Coverings

We compute the individual polygon coverings using the S2 library. Note that our approach does not depend on S2 and, in fact, works with any other quadtree-based hierarchical grid in which each (implicit) quadtree node [71] corresponds to a geographical area (space partitioning). For our approach to work, each quadtree node needs to be *uniquely identifiable* with a bit sequence

that represents the path to the given node starting from the root. Thereby, any (consistent) enumeration scheme (e.g., the Hilbert space-filling curve used by S2 or the Z curve used by Roth [177]) of the four quadrants is valid. To store these encoded node identifiers in a trie, we require the identifiers of child nodes to share a common prefix with their parent node.

Face Nodes

Since our implementation uses S2, which projects points on Earth onto a surrounding cube, we need to maintain up to six radix trees (one for each face). Using the first three bits of the query cell id, we select the appropriate radix tree.

Index Probing

The probe (filter) phase is the performance-critical part of our approach. We therefore parallelize this phase to accelerate lookups in the radix tree. Individual processing threads fetch batches of 16 tuples at a time and synchronize using an atomic counter.

PIP Test

In the refinement phase, we use S2's PIP test, which implements the ray tracing algorithm (cf. [163] for performance numbers).

3.4 EXPERIMENTAL EVALUATION

In this section, we present a thorough experimental analysis of our point-polygon join algorithms. We use taxi data from NYC, which we join with different polygonal regions of NYC, such as neighborhoods. We also experiment with geo-tagged Twitter data from different cities. Besides these (skewed) real-world point datasets, we experiment with (uniform) synthetic point data. We focus our experiments on the probe phase of the join (probing points against a pre-built polygon index). For completeness, we also report build times.

Our evaluation is structured into three parts: First, we evaluate the performance and space consumption of our approximate algorithm with different data structures, including ACT, a B-tree, and a sorted vector. We demonstrate that for a city like NYC (with its 289 neighborhoods), an approximate index with very high precision (<4 m precision bound) easily fits into the main memory of a single machine and, in the case of ACT, allows for very high probe performance (>50 M points/s per CPU core). We think that this is a good fit with the city-centric model of mobility companies (e.g., Uber, DriveNow [56]).

Table 6: Metrics of the NYC polygon datasets.

	no. of polygons	avg. no. of vertices
boroughs	5	662
neighborhoods	289	29.6
census	39184	12.5

We show that ACT outperforms other physical representations by a large margin, while being more space-efficient in many cases. Second, we evaluate our accurate algorithm and show that it benefits greatly from true hit filtering. We compare it against other filter and refine approaches, including an R-tree on the polygons' MBRs, a geospatial index by Google, and PostgreSQL (PostGIS). We demonstrate that the high precision of our index can be further improved by training it with historical data points. Third, we show that both our algorithms are competitive with state-of-the-art GPU approaches.

3.4.1 Infrastructure

We use a server-class machine that is equipped with two 14-core Intel Xeon E5-2680 v4 CPUs and 256 GiB DDR4 RAM. All CPU-based approaches are implemented in C++ and compiled with GCC version 5.4.0 with `O3` and `march=core-avx2` flags. We conduct the experiments on a single socket to eliminate NUMA effects. For the comparison against the GPU join algorithms, we use these Amazon Web Services (AWS) instances [15]:

C5.4XLARGE 16 vCPUs, USD 0.68/hour

G3S.XLARGE NVIDIA Tesla M60 GPU, USD 0.75/hour

3.4.2 Datasets and Queries

We use 1.23 B points (pick-up locations) from the NYC yellow taxi dataset (years 2009 to 2016), which is publicly available in CSV format [196]. For each point, we load its lat/lng coordinate and convert it to an `S2Point` [179] (which represents a point on the unit sphere as a 3D vector of doubles) and to an `S2CellId` (an 8 byte value, cf. Section 3.2) prior to performing any experiments. We maintain one `std::vector` of `S2Points` and another one storing the corresponding cell ids. We join these points against NYC's boroughs (5 polygons) [27], neighborhoods (289 polygons) [150], and census blocks (39,184 polygons) [33]. All three polygon datasets cover approximately the same area. While there are only five boroughs, their polygons are significantly more complex. Table 6 summarizes the metrics of the polygon datasets.

In addition, we use geo-tagged tweets collected from Twitter's live public feed over a period of five years. From these, originally over 2.29 B tweets spread across the entire US, we extract four point datasets based on the MBRs of NYC, Boston (BOS), Los Angeles (LA), and San Francisco (SF), consisting of 83.1 M, 13.6 M, 60.6 M, and 9.57 M points, respectively. We join these points against the corresponding neighborhood polygons: NYC (289), BOS (42), LA (160), and SF (117). Since we extract the points using the MBR of the entire polygon dataset and not the individual neighborhood polygons, there are points that do not join with any polygon.

We also generate synthetic point datasets, uniformly distributed within the MBR of the respective polygon dataset.

We focus our experimental evaluation on the probe phase and simply count the number of points per polygon instead of materializing the join result. To avoid any contention in the multi-threaded experiments, we maintain thread-local counters that we aggregate in the last step. Since we are focusing on the case of static polygons, the reported throughput times reflect the time to compute the counts using an *existing* (pre-built) polygon index. We report the time it takes to build the polygon index separately. However, we would like to point out that we did not optimize the build phase.

3.4.3 Polygon Approximations

Our default configuration for computing the individual polygon coverings is as follows: max covering cells = 128, max covering level = 30, max interior cells = 256, and max interior level = 20.

3.4.4 Approximate Join

We first analyze the performance and space consumption of our approximate algorithm. In all of the following experiments, we first build super coverings (sets of cell/value pairs, cf. Section 3.3) and then index them with different data structures.

Super Covering Construction

Table 7 shows different metrics of the super coverings for the three polygon datasets with 60 m, 15 m, and 4 m precision. With each cell occupying 64 bits, the largest super covering (census 4 m, 39.8 M cells) amounts to 304 MiB of raw key data and another 304 MiB for the values (64 bit tagged entries, cf. Section 3.3). Given that most cells reference fewer than three polygons, most polygon references are inlined, which keeps the lookup table small. While the computation of the individual coverings is parallelized over the number of polygons, the construction of the super covering is performed serially.

Table 7: Metrics of three super coverings with various precisions.

precision [m]	60	15	4	60	15	4	60	15	4
# cells [M]	0.09	1.32	20.9	0.16	0.98	14.0	8.50	8.97	39.8
lookup table [MiB]	0.00	0.00	0.00	0.01	0.01	0.01	1.33	1.33	1.41
build individ. coverings [s]	0.11	0.98	16.0	0.07	0.19	1.54	0.96	1.01	3.08
build super covering [s]	0.10	0.94	15.2	0.17	0.81	10.5	11.6	11.8	37.7

Table 8: Metrics of the different data structures (4 m precision).

super cov.	boroughs (20.9 M cells)					neighborhoods (14.0 M cells)					census (39.8 M cells)				
index	ACT1	ACT2	ACT4	GBT	LB	ACT1	ACT2	ACT4	GBT	LB	ACT1	ACT2	ACT4	GBT	LB
size [MiB]	328	198	173	359	319	224	138	143	240	214	624	421	1234	684	608
build [s]	2.11	1.46	1.06	1.39	-	1.36	0.98	0.69	0.85	-	4.00	3.11	2.80	2.85	-

Data Structures

We essentially need to map cell ids (64 bit integers) to tagged entries (64 bit values). A tagged entry either contains up to two polygon references or an offset into a lookup table. The lookup table is the same among all data structures that we evaluate. The data structure needs to support prefix lookups: given a 64 bit lookup key (the cell id of a query point), find the cell in the super covering (recall that it only contains non-overlapping cells) that shares a common prefix with the lookup key (if such a cell exists). We analyze ACT with three different fanouts: 2, 4, and 8 bits per radix level, which corresponds to 1, 2, and 4 quadtree levels, respectively. Recall that one quadtree level is encoded with two bits. We therefore refer to these three variants as ACT1, ACT2, and ACT4. As competitors we use a B-tree implementation by Google [44] (GBT) and a binary search on a sorted vector implemented with `std::lower_bound` (LB). For GBT, we use a (target) node size of 256 bytes, which turned out to be the most query-efficient configuration. The vector stores pairs of cell ids and tagged entries. We have also experimented with the STX B+-tree [189] but do not include it in this section as its lookup performance is very similar to that of GBT.

The performance of our approximate algorithm is dominated by the costs of the ACT node accesses and the aggregation (count). To better understand the results, we therefore first analyze the space consumption of ACT and compare it with GBT and the sorted vector. Table 8 shows size and build time (single threaded) of the different data structures on the super coverings introduced above (4 m precision only). In many cases, ACT consumes less space than the sorted vector (LB). Due to the high density of the cell ids, ACT is more space-efficient with higher fanouts, except for census where ACT4 consumes the most space: Like for all datasets, ACT4 has fewer (but larger) nodes than ACT1 and ACT2. However, in this case, its nodes are very sparsely populated compared to those of ACT1 and ACT2. The reason is that ACT4's nodes cover too much space for the relatively small census cells. All 4 m indexes exceed the 35 MiB L3 cache of our evaluation machine. Note that there is no additional build time for LB, since the super covering contains cell id/tagged entry pairs already sorted by cell id.

Single-Threaded Throughput

For this experiment, we compute a super covering with a 4 m precision bound on the three NYC polygon datasets and store it in the different data structures introduced above. We then join the full taxi dataset (all 1.23 B points) against each of these indexes and report the throughput in M points/s (cf. Figure 29).

ACT clearly dominates the B-tree and the binary search on the sorted vector, especially in its highest fanout configuration (ACT4). A higher fanout means

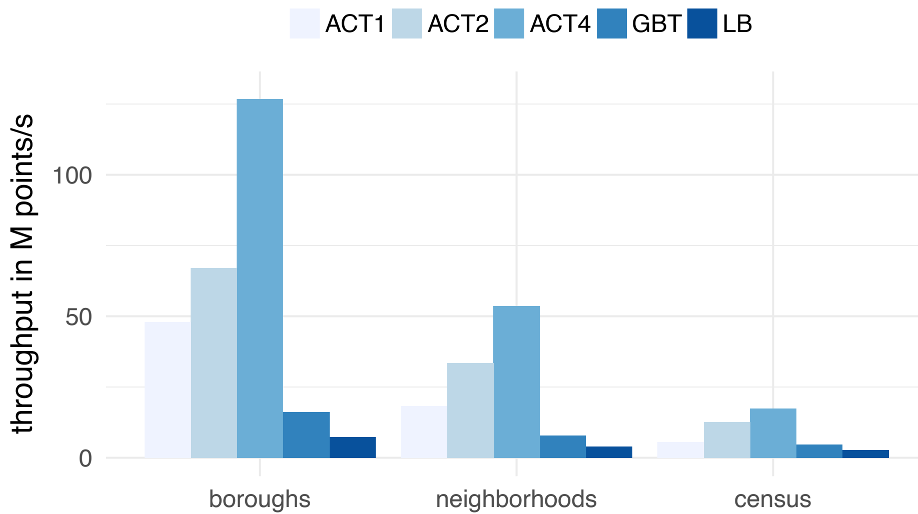


Figure 29: Single-threaded throughput of our approximate algorithm with different data structures (4 m precision).

Table 9: Speedups of lookups in smaller (more coarse-grained) over larger (more fine-grained) polygon datasets for different data structures (b = boroughs, n = neighborhoods, c = census).

	b over n	b over c	n over c
ACT1	**2.63**×	**8.63**×	**3.28**×
ACT2	2.00×	5.33×	2.66×
ACT4	2.36×	7.29×	3.08×
GBT	2.05×	3.51×	1.71×
LB	1.83×	2.63×	1.44×

that we consume more bits of the lookup key per tree level and thus require fewer node accesses (i.e., need to traverse fewer levels) to find a key (an indexed cell). With ACT4 for example, we consume 8 bits per tree level and thus need at most $64/8 = 8$ node accesses. Since we reduce the tree height further by storing a common prefix at the root level (cf. Section 3.3), ACT requires even fewer node accesses (e.g., at most five with 4 m precision).

Another insight is that ACT benefits the most from the larger (coarser-grained) cells in the smaller polygon datasets as shown in Table 9. Going from the most fine-grained census dataset (39,184 polygons) to the most coarse-grained boroughs dataset (5 polygons), GBT's lookup performance improves by 3.51×, while ACT1's increases by 8.63×. The reason for ACT's large gain is that larger cells are indexed higher up in the radix tree and are thus found sooner. GBT, in contrast, does not benefit from these larger cells, which might as well be stored in the leaf nodes of the B-tree. GBT's performance gain comes from the smaller number of cells used for indexing the boroughs dataset and the result-

ing smaller B-tree (i.e., fewer branch and cache misses per point). Likewise, the binary search on the sorted vector (LB) is only affected by the number of cells and not their granularity.

Different Precisions

Next, we vary the precision of the indexed super covering. We perform this experiment using the medium size neighborhoods dataset. Figure 30 shows the throughput numbers for the different data structures. While GBT's and LB's performance decreases by 33.4% and 39.4%, respectively from 60 m to 4 m, ACT4's performance is hardly affected (-5.73%) by the larger number of cells of the more precise super covering. Compared to the 60 m covering, the more precise coverings contain a larger number of small cells (in the boundary areas of the polygons). Query points are unlikely to hit these cells in contrast to the large (more coarse-grained) cells, which are indexed in the upper (cached) ACT nodes (due to their shorter cell ids). ACT1 and ACT2 are more affected by the precision increase (-27.8% and -17.9%, respectively). The reason is that the added small cells have a stronger effect on the depths of these trees. While the average node depth for ACT4 only increases from 2.83 to 2.97 (+4.95%) from 60 m to 4 m respectively, the same metric increases from 10.8 to 14.6 (+35.2%) for ACT1. Although—as already stated above—the new small cells are unlikely to be hit, they still cause a performance hit for lower fanouts.

ACT4's throughput is similar for 15 m and 4 m (-4.15%) because its structure is identical for both precisions. In both cases, it has 70,786 nodes occupying 143 MiB. The only difference is the nodes' structure: Due to the more fine-grained cell approximation, the average node occupancy (measured in terms of occupied slots) of ACT4 at tree level 3 decreases from 88.2% (60 m) to 85.2% (4 m). The occupancies of all other levels are the same. This lower occupancy for 4 m saves some aggregations (for updating the polygon hit counts), causing a slightly higher performance.

In summary, the impact of precision on query performance is less significant for ACT than for the other data structures.

Multi-Threaded Throughput

In this experiment, we study the lookup performance of the different data structures with an increasing number of threads on the neighborhoods dataset with a 4 m precision bound. We use up to 28 threads, which matches the number of hyperthreads of a single NUMA node of our evaluation machine. Figure 31 shows the speedups over single-threaded execution. Up to 8 threads, all index structures scale almost linearly (speedup of around 7× in all cases). This is what we would expect for immutable data structures.

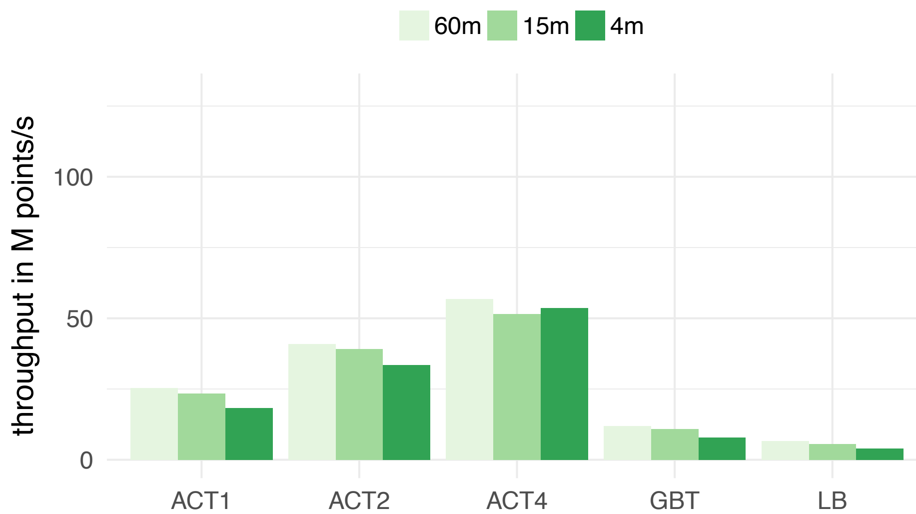
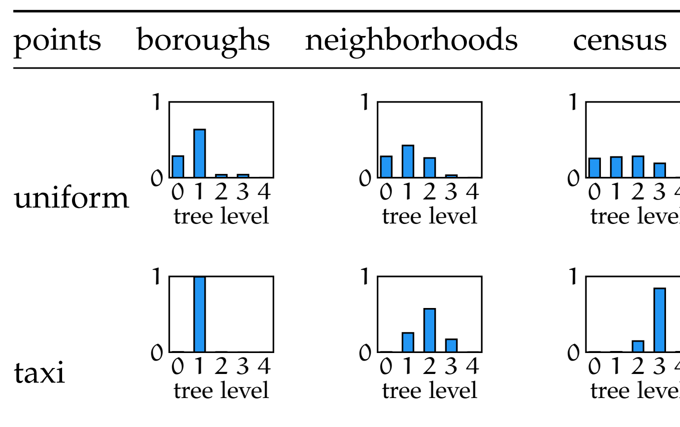


Figure 30: Single-threaded throughput of our approximate algorithm with different precisions and data structures (neighborhood polygons).

Table 10: Distribution of the tree traversal depth (ACT4 with 4 m precision).



points	boroughs	neighborhoods	census
uniform	tree level (0 1 2 3 4)	tree level (0 1 2 3 4)	tree level (0 1 2 3 4)
taxi	tree level (0 1 2 3 4)	tree level (0 1 2 3 4)	tree level (0 1 2 3 4)

The fact that an oversubscription of cores (hyperthreading) has a positive performance impact suggests that the lookup is bound by memory access latencies (having more threads than physical cores can hide these latencies).

Synthetic Points

To show the general applicability of our approach, we also experiment with synthetic point data. We generate 100 M points uniformly distributed within the MBR of the respective (NYC) polygon dataset. Table 10 shows the probability distribution of the number of search steps during the tree traversal for the synthetic and the taxi point dataset. As expected, the distribution for the uniform data is skewed towards the root. That is because the larger cells (which are more likely to be hit) are indexed closer to the root. The distribution for the taxi data depends on the polygon dataset. For boroughs, most traversals

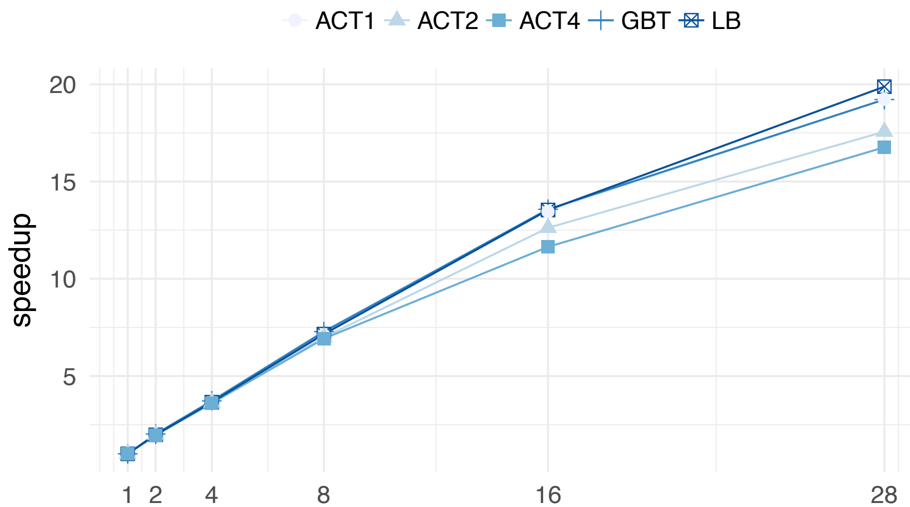


Figure 31: Multi-threaded throughput of our approximate algorithm with different data structures (neighborhood polygons, 4 m precision).

Table 11: Performance counters per point (neighborhoods, 4 m precision).

points	uniform					taxi				
index	ACT1	ACT2	ACT4	GBT	LB	ACT1	ACT2	ACT4	GBT	LB
cycles	154	99.8	71.3	415	569	172	93.8	56.4	416	817
instruct.	214	121	82.4	486	927	202	121	81.3	393	564
br. miss.	1.06	1.04	0.88	5.32	8.38	0.96	0.83	0.48	7.06	10.8
LLC miss.	0.29	0.23	0.18	0.70	1.89	0.22	0.17	0.15	0.29	0.37

end at tree level 1, while for census, points mostly hit small cells indexed in tree level 3.

Figure 32 shows the single-threaded throughput for the different data structures with the uniform point data. ACT achieves the highest throughput, with ACT4 again being the most query-efficient configuration. The absolute numbers, however, are lower than for the (real-world) taxi data: ACT4's throughput decreases by 65.2%, 26.8%, and 3.11% for boroughs, neighborhoods, and census, respectively.

The reason for this slowdown is simple: The synthetic point data is uniformly distributed, which leads to more branch and cache misses (cf. Table 11 for performance counters on neighborhoods). In contrast, the real-world taxi data is highly clustered with the majority of points located in Manhattan (>90%) and around the airports. For boroughs (not shown in Table 11), ACT4 endures 0.79 and 0.01 branch misses per point for the synthetic and the taxi points, respectively. This is the main cause of the 65.2% performance drop mentioned above.

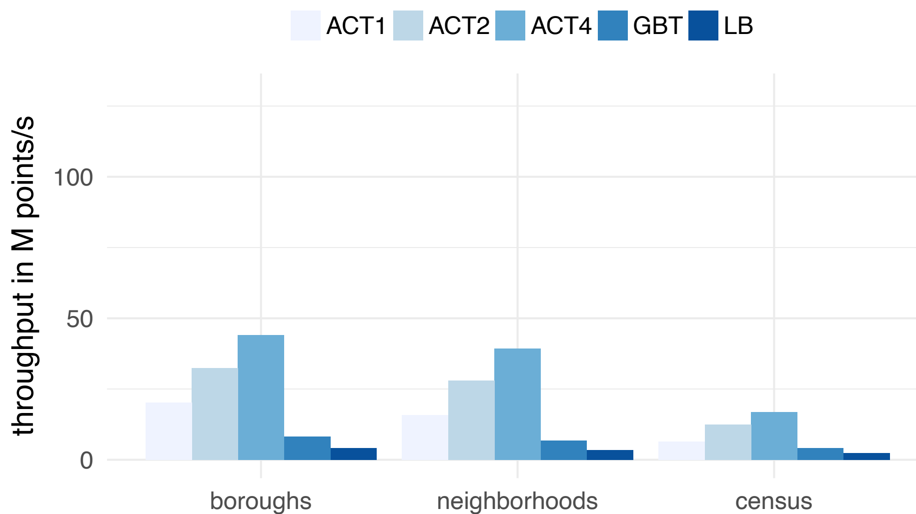


Figure 32: Single-threaded throughput of our approximate algorithm (4 m precision) with uniform point data.

Twitter Data

Next, we analyze the performance of our approach on the four Twitter datasets and the corresponding neighborhood polygons (cf. Figure 33). The numbers are similar across the different cities, with the highest throughput achieved for BOS with its only 42 neighborhood polygons. Next comes SF followed by LA and NYC, for which the throughput is very close to what we obtained with the taxi data (cf. Figure 29. In fact, with a 4 m precision, ACT4 achieves a single-threaded throughput of 52.1 M points/s, which is almost the same as the 53.6 M points/s on the taxi data. Similarly to the taxi points, the tweets are clustered, with certain areas having more tweeting activity than others. In contrast, with uniform point data, ACT4 only achieved 39.3 M points/s. This confirms that our approach benefits from the skewed distribution of real-world data. For all four cities, the numbers are (again) hardly affected by the precision.

3.4.5 Accurate Join

We now evaluate our accurate algorithm, which eliminates false positives in an additional refinement phase. We demonstrate that our index benefits significantly from true hit filtering and that index training with historical data can further improve its effect.

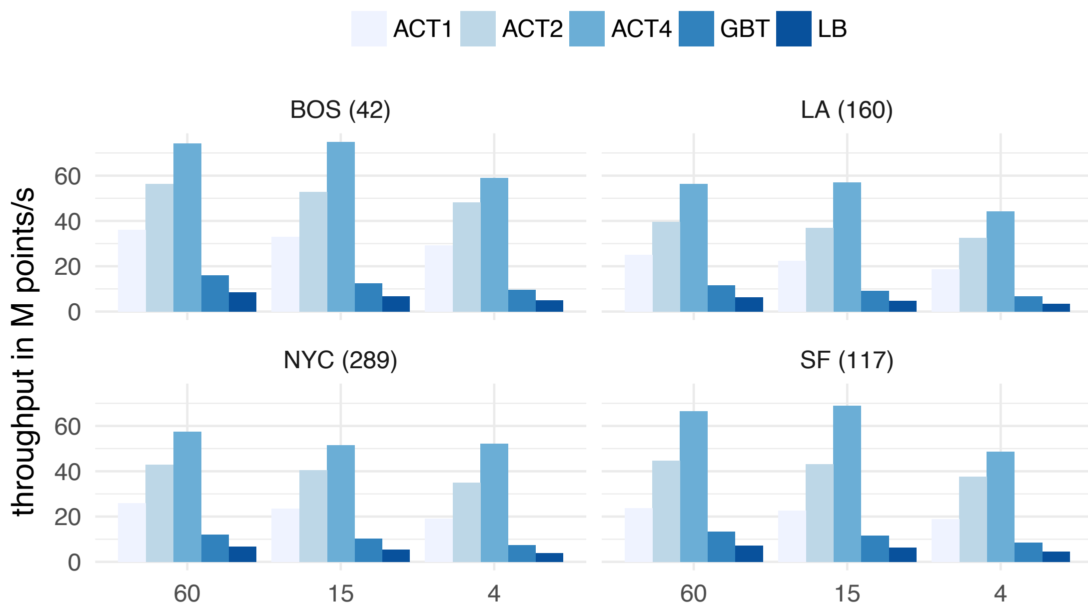


Figure 33: Single-threaded throughput of our approximate algorithm (Twitter datasets, polygon counts in brackets).

Competitors

We compare against the boost R-tree (1.6.0) [26] on the polygons' MBRs (RT), Google's S2ShapeIndex [180] (SI), and PostgreSQL 9.6.1 (PostGIS 2.3.1) [170] with a GiST index on polygons (PG). Our algorithm and the R-tree both use the same PIP test implementation (cf. Section 3.3.4). SI also uses that implementation, however, restricts the test to a subset of edges of the polygon in question. This is achieved by using a hierarchical grid approximation of polygons, and internally mapping grid cells (64 bit S2CellIds) to polygon edges using a B-tree. This hierarchical grid approximation is much more coarse-grained than our super covering, given its higher focus on build time than on query performance (compared to our approach). SI allows the maximum number of edges per cell to be configured, essentially controlling the granularity of the employed grid approximation. We evaluate SI with its default configuration of 10 edges (SI10) and 1 edge per cell (SI1). Note that SI1 is the most fine-grained configuration possible. SI also employs true hit filtering (cf. Section 3.3) to avoid PIP tests, but in a much less effective way than ours (due to its coarser-grained grid). Furthermore, SI does not offer an approximate version. For the R-tree, we use the splitting strategy `rstar` with at most 8 elements per node which performs best in all workloads.

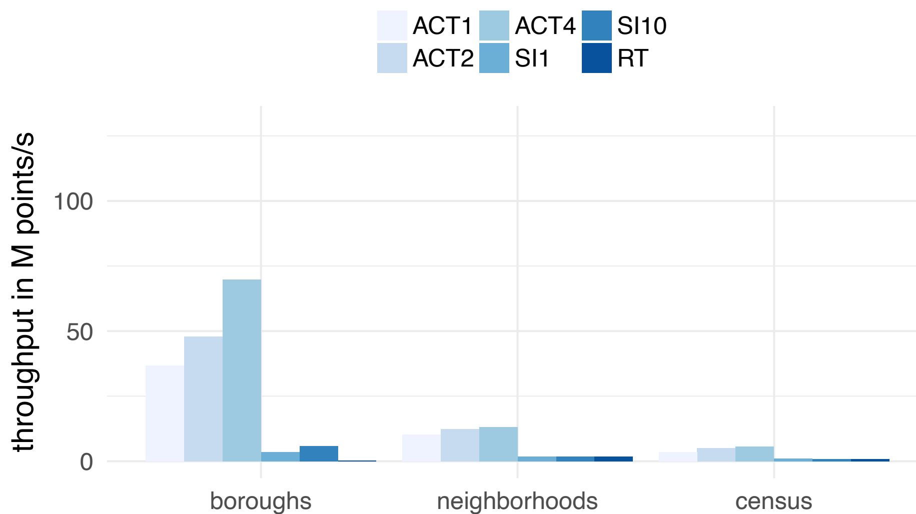


Figure 34: Single-threaded throughput of our accurate algorithm (with different ACT fanouts) compared to S2ShapeIndex (with 1 and 10 edges per cell) and the R-tree.

Taxi Data

For this experiment, we compute coarse-grained super coverings that do not guarantee a certain precision, and instead fall back on a refinement phase for candidate hits. Here, the resolution of a super covering is determined by our default configuration for computing individual polygon coverings introduced earlier (cf. Section 3.4). Thus, these super coverings consist of much fewer cells than those guaranteeing a certain precision. For example, the approximation for the neighborhoods dataset now only consists of 98,687 cells (ACT4 size: 25.9 MiB) compared to the 13.2 M cells (ACT4 size: 143 MiB) needed to guarantee a 4 m precision. For this dataset, SI1, SI10, and RT consume 1.20 MiB, 0.23 MiB, and 27.9 KiB, respectively.

Figure 34 shows the single-threaded throughputs for the accurate join. ACT4 achieves the highest performance for all three datasets. For the medium size neighborhoods dataset, it outperforms SI1 by 6.96×, followed by SI10, which is only 7.41% slower than SI1. For census, ACT4 still outperforms SI1 by 5.79×. RT has the lowest numbers with 0.21, 1.77, and 0.79 M points/s for boroughs, neighborhoods, and census, respectively. The reason for its slow performance for boroughs is as follows: The complexity of each PIP test (ray-tracing algorithm) is linear with the size (number of edges) of the polygon. Since the boroughs are complex polygons with many edges, the PIP tests in the refinement phase are very expensive. Here, our algorithm shines since it can identify most join partners in the filter phase and only enters the refinement phase for 0.1% of the points. As a point of reference, PG achieves 0.39, 1.09, and 0.69 M points/s for boroughs, neighborhoods, and census, respectively (because we

Table 12: Speedups of single-threaded lookups when training ACT4 with an increasing number of historical data points (over untrained ACT4).

no. of train. points	boroughs	neighborhoods	census
100 K	1.25×	1.56×	1.16×
500 K	1.40×	2.00×	1.40×
1 M	1.44×	2.18×	1.53×

Table 13: Effect of training the index with 1 M historical data points (STH = solely true hits).

metric	boroughs	neighborhoods	census
STH (%)	99.9 → 99.9	87.2 → 97.7	72.2 → 88.7

use all hyperthreads on our evaluation machine, PG's numbers are not directly comparable and are excluded from the plot). Similar to RT, PG is affected by the complex boroughs polygons.

Index Training

As readers may have noticed, there is a large performance gap between our approximate and our accurate algorithm. For example, ACT4 (accurate) is 75.3% slower than its approximate counterpart (with 4 m precision) on the taxi data/neighborhoods join. The reason is the expensive PIP tests needed to compute an accurate result.

We now show how to narrow this performance gap. The idea is to reduce the likelihood for PIP tests by training the index with historical data points (cf. Section 3.3.3). In other words, we increase the precision of the index by making it more fine-grained in areas where we expect more points. One effect this has is that the size of the area covered by (expensive) boundary cells will decrease. We train the index with taxi points sampled from the year of 2009 and only use the points from 2010 to 2016 for the join. Table 12 shows the performance impact. With 100 K training points, ACT4's performance improves by 1.56× for neighborhoods and increases further to 2.18× with 1 M points (due to a 84.0% reduction in the number of PIP tests). The size of ACT4 only increases from 25.9 MiB (untrained) to 28.0, 34.8, and 44.3 MiB when trained with 100 K, 500 K, and 1 M historical data points, respectively. In absolute terms, ACT4 trained with 1 M points achieves a throughput of 29.1 M points/s for neighborhoods and thus narrows the performance gap to its approximate counterpart (with 4 m precision) from 75.3% to 45.7% while consuming 68.9% less space. This shows that a trained accurate index is a good alternative to our approximate indexes when main memory is sparse. Table 13 shows the effect of true hit filtering when training the index with 1 M training points. The metric `solely`

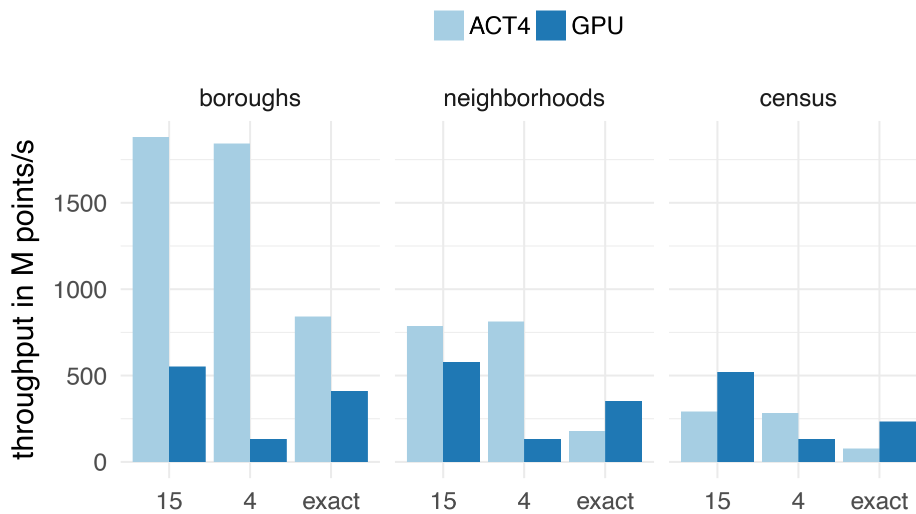


Figure 35: Throughput of ACT4 (16 threads) compared to the two GPU algorithms on AWS (GPU = Bounded Raster Join for 15 m and 4 m and Accurate Raster Join for exact).

`true hits` (STH) indicates the percentage of points that skipped the expensive refinement phase, which is clearly above 70% in all cases (even without training). Training the index significantly improves STH for neighborhoods and census.

3.4.6 Comparison with GPU Algorithms

Finally, we compare our approximate and accurate (untrained ACT) algorithms against state-of-the-art GPU counterparts [215]. The GPU approaches leverage the graphics rendering pipeline, and in particular the rasterization operation, which converts a polygon into a collection of (equi-sized) pixels. Similar to our approach, the GPU join also comes in two variants: Bounded Raster Join (BRJ), which guarantees a user-defined precision by appropriately scaling the rendering resolution, and Accurate Raster Join (ARJ), which performs PIP tests for points falling on the pixels forming the boundaries of the polygons. To enable a fair comparison, we do not consider any preprocessing times on the polygons (such as triangulation time). Note that the preprocessing time for the GPU join is minimal. In fact, it is designed for computing the join on-the-fly without a priori knowledge of the polygonal regions.

We now compare the throughput of both approaches on two similarly priced AWS machines (cf. Infrastructure in Section 3.4). Figure 35 shows the results of joining 612 M taxi rides with the NYC polygon datasets. While our approximate algorithm is again hardly affected by the precision (15 m vs. 4 m), BRJ takes a significant performance drop. The reason for BRJ's slowdown is simple: Once the required resolution is higher than what is natively supported by the

GPU, it needs to split the scene and perform more rendering passes. This is essentially related to the fact that BRJ relies on a uniform grid. On the contrary, BRJ is barely affected by the polygon datasets, while our approximate algorithm is. The reason is again related to the granularity of the grid: With the more fine-grained census dataset, we need to traverse more tree nodes (as the cells that approximate the polygons are smaller), while the rendering resolution in BRJ depends only on the size of the bounding box of the polygon dataset and the precision. With exact results, our approach outperforms ARJ for boroughs, while ARJ takes the crown for neighborhoods and census.

3.5 RELATED WORK

3.5.1 Spatial Join Techniques

The point-polygon join is one of the core operations in spatial databases, and, a large body of related work on algorithmic techniques [87] is available accordingly.

Naturally, we are not the first to index polygons using raster approximations. Early on, Orenstein [159] proposed decomposing single polygons into multiple cells. Later, Brinkhoff et al. [31] proposed true hit filtering in the form of maximum enclosed rectangles and circles, allowing the refinement phase to be skipped in many cases. Zimbrao et al. [225] followed up on this approach by using raster approximations in the form of uniform grids, thereby improving selectivity. Kothuri et al. [93] recursively divide the MBR of a polygon into four cells until a certain granularity is reached, identify interior cells, and index them in an R-tree to skip refinement checks. The primary goal was to minimize I/O, an important performance factor for disk-based systems. In contrast to these early works on true hit filtering and also to the recent proposal by Tzirita Zacharatou et al. [215], we use a quadtree-based (multi-resolution) grid that can be very coarse-grained in interior and very fine-grained in boundary areas.

Research has, however, also been performed on true hit filtering with quadtree-based rasterizations, including work in Oracle Spatial [94] and Microsoft SQL Server [61]. In both of these works, *individual* polygons are approximated using a set of multi-resolution grid cells. These grid cells are enumerated using one-dimensional cell identifiers and stored in a B-tree. In contrast, we *holistically* approximate and index an entire set of polygons and store these (in our case duplicate-free) cell identifiers in a novel radix tree (ACT), which is more query-efficient than a B-tree. Additionally, these existing approaches neither offer an approximate mode nor allow the accurate index to be trained with historical data points to improve query performance.

To decrease the probability of false matches, [184] improves the precision of MBRs by clipping away empty space that is concentrated around the MBR cor-

ners. In contrast to our work, [184] uses the classical filter and refine evaluation strategy.

Related to our approximate algorithm is work by Azevedo et al. [17] that provides precision estimates for approximate polygon-polygon joins using a less space-efficient single-resolution grid. Tzirita Zacharatou et al. [215] propose a similar precision bound to ours but also use a single-resolution grid (cf. Section 3.4.6 for a comparison).

The PH-tree [221] is another example of a trie data structure that indexes multi-dimensional data. In contrast to ACT, it only indexes points, not higher-level grid cells.

3.5.2 Systems

Several database systems support geospatial joins. PostGIS [170], a geospatial extension to PostgreSQL [171], uses an R-tree implemented on top of GiST [82] for indexing geospatial objects. In recent years, various spatial data management systems based on Hadoop [4, 59] and Spark [213, 210, 194] have emerged. In contrast to our work, most of these systems rely on offline partitioning of the data points.

3.5.3 Modern Hardware

Most work on using modern hardware for geospatial joins focuses on GPU offloading [224, 212, 1, 215, 55] while [39] proposes a GPU-accelerated end-to-end spatial system.

In [118, 120], we describe a novel approach to reduce control flow divergence on AVX-512 platforms to further increase ACT's lookup performance.

3.6 CONCLUSIONS

We have presented two point-polygon join algorithms that use a multi-resolution grid indexed in a query-efficient radix tree. We have transformed a traditionally compute-intensive problem into a memory-intensive one. We have shown that it is possible to refine the index up to a user-defined precision and identify all join partners in the filter phase. We have demonstrated that the accurate version of our algorithm can adapt to the expected point distribution. We have also shown that our approach outperforms existing CPU-based joins by up to two orders of magnitude and can compete with dedicated GPU implementations.

4 LEARNED CARDINALITIES

Excerpts of this chapter have been published in [100, 106, 99].

4.1 INTRODUCTION

Query optimization is fundamentally based on cardinality estimation. To be able to choose between different plan alternatives, the query optimizer must have reasonably good estimates for intermediate result sizes. It is well known, however, that the estimates produced by all widely-used database systems are routinely wrong by orders of magnitude—causing slow queries and unpredictable performance. The biggest challenge in cardinality estimation are join-crossing correlations [124, 128]. For example, in the Internet Movie Database (IMDb), French actors are more likely to participate in romantic movies than actors of other nationalities.

The question of how to better deal with this is an open area of research. One state-of-the-art proposal in this area is Index-Based Join Sampling (IBJS) [127] that addresses this problem by probing qualifying base table samples against existing index structures. However, like other sampling-based techniques, IBJS fails when there are no qualifying samples to start with (i.e., under selective base table predicates) or when no suitable indexes are available. In such cases, these techniques usually fall back to an "educated" guess—causing large estimation errors.

The past decade has seen the widespread adoption of machine learning (ML), and specifically neural networks (deep learning), in many different applications and systems. The database community also has started to explore how machine learning can be leveraged within data management systems. Recent research therefore investigates ML for classical database problems like parameter tuning [6], query optimization [140, 161, 112], and even indexing [110].

We argue that machine learning is a highly promising technique for solving the cardinality estimation problem. Estimation can be formulated as a supervised learning problem, with the input being query features and the output being the estimated cardinality. In contrast to other problems where machine learning has been proposed like index structures [110] and join ordering [140], the current techniques based on basic per-table statistics are not very good. In other words, an estimator based on machine learning does not have to be perfect, it just needs to be better than the current, inaccurate baseline. Fur-

thermore, the estimates produced by a machine learning model can directly be leveraged by existing, sophisticated enumeration algorithms and cost models without requiring any other changes to the database system.

In this work, we propose a deep learning-based approach that learns to predict (join-crossing) correlations in the data and addresses the aforementioned weak spot of sampling-based techniques. Our approach is based on a specialized deep learning model called multi-set convolutional network (MSCN) allowing us to express query features using sets (e.g., both $(A \bowtie B) \bowtie C$ and $A \bowtie (B \bowtie C)$ are represented as $\{A, B, C\}$). Thus, our model does not waste any capacity for memorizing different permutations (all having the same cardinality but different costs) of a query's features, which results in smaller models and better predictions. The join enumeration and cost model are purposely left to the query optimizer. We evaluate our approach using the real-world IMDb dataset [124] and show that our technique is more robust than sampling-based techniques and even is competitive in the sweet spot of these techniques (i.e., when there are many qualifying samples). This is achieved using a (configurable) low footprint size of about 3 MiB (whereas the sampling-based techniques have access to indexes covering the entire database). These results are highly promising and indicate that ML might indeed be the right hammer for the decades-old cardinality estimation job. Our code and the workloads used for evaluation are released on GitHub [122]. Further, we discuss another application of our model, which is predicting the number of unique values in a (combination of) columns (cf. Section 4.4). We experimentally show the poor performance of state-of-the-art approaches for estimating group-by queries, adapt our learning-based approach to this problem, and show that it provides higher accuracy. Finally, we describe a prototype that demonstrates the end-to-end training and cardinality estimation process and thereby sketch a possible integration into the query optimizer (cf. Section 4.5).

The remainder of this chapter is structured as follows: Section 4.2 describes our approach and Section 4.3 presents the evaluation with real-world data. Section 4.4 outlines how our approach can be applied to group-by queries. Section 4.5 demonstrates the end-to-end training and estimation process. Section 4.6 discusses the limitations of our approach and possible ways forward. Section 4.7 describes related work and is followed by conclusions in Section 4.8.

4.2 APPROACH

From a high-level perspective, applying machine learning to the cardinality estimation problem is straightforward: after training a supervised learning algorithm with query/output cardinality pairs, the model can be used as an estimator for other, unseen queries. There are, however, a number of challenges

that determine whether the application of machine learning will be successful: the most important question is how to represent queries ("featurization") and which supervised learning algorithm should be used. Another issue is how to obtain the initial training dataset ("cold start problem"). In the remainder of this section, we first address these questions before discussing a key idea of our approach, which is to featurize information about materialized samples.

4.2.1 Set-Based Query Representation

We represent a query $q \in Q$ as a collection (T_q, J_q, P_q) of a set of tables $T_q \subset T$, a set of joins $J_q \subset J$ and a set of predicates $P_q \subset P$ participating in the specific query q. T, J, and P describe the sets of all available tables, joins, and predicates, respectively.

Each table $t \in T$ is represented by a unique *one-hot* vector v_t (a binary vector of length $|T|$ with a single non-zero entry, uniquely identifying a specific table) and optionally the number of qualifying base table samples or a bitmap indicating their positions (sample bitmap). Similarly, we featurize joins $j \in J$ with a unique one-hot encoding. For predicates of the form (col, op, val), we featurize columns col and operators op using a categorical representation with respective unique one-hot vectors, and represent val as a normalized value $\in [0, 1]$, normalized using the minimum and maximum values of the respective column.

Applied to the query representation (T_q, J_q, P_q), our MSCN model (cf. Figure 36) takes the following form:

$$\text{Table module:} \quad w_T = \frac{1}{|T_q|} \sum\nolimits_{t \in T_q} \text{MLP}_T(v_t)$$

$$\text{Join module:} \quad w_J = \frac{1}{|J_q|} \sum\nolimits_{j \in J_q} \text{MLP}_J(v_j)$$

$$\text{Predicate module:} \quad w_P = \frac{1}{|P_q|} \sum\nolimits_{p \in P_q} \text{MLP}_P(v_p)$$

$$\text{Merge \& predict:} \quad w_{\text{out}} = \text{MLP}_{\text{out}}([w_T, w_J, w_P])$$

Figure 37 shows an example of a featurized query.

4.2.2 Model

Standard deep neural network architectures such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), or simple multi-layer perceptrons (MLPs) are not directly applicable to this type of data structure, and would require *serialization*, i.e., conversion of the data structure to an ordered sequence of elements. This poses a fundamental limitation, as the model would have to spend capacity to learn to discover the symmetries and structure

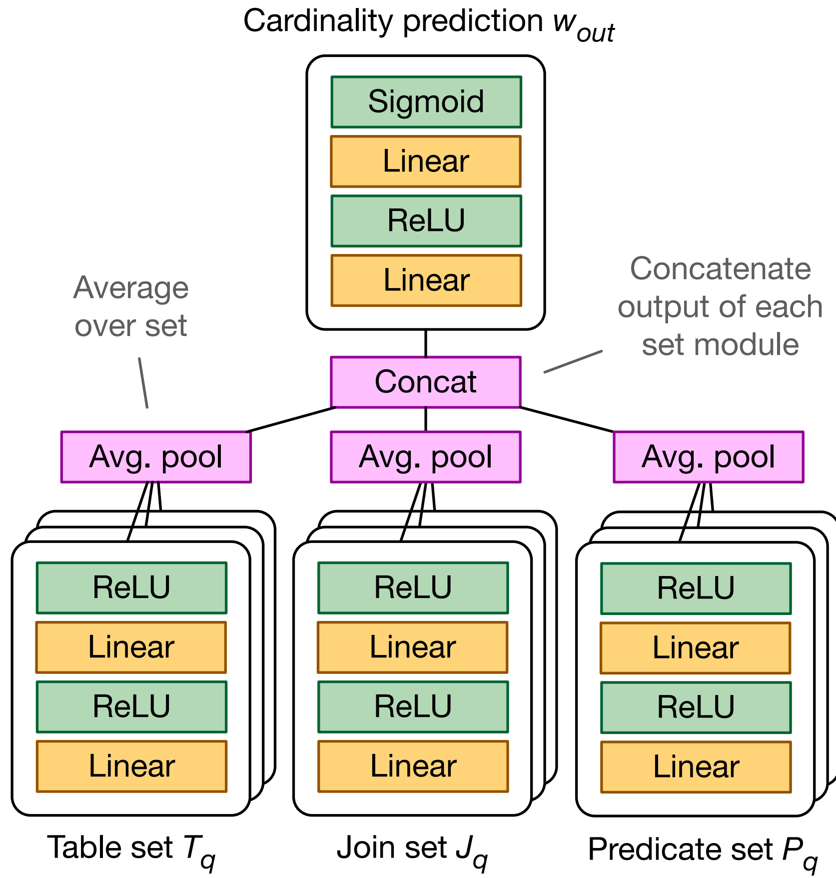


Figure 36: Architecture of our multi-set convolutional network. Tables, joins, and predicates are represented as separate modules, comprised of one two-layer neural network per set element with shared parameters. Module outputs are averaged, concatenated, and fed into a final output network.

of the original representation. For example, it would have to learn to discover boundaries between different sets in a data structure consisting of multiple sets of different size, and that the order of elements in the serialization of a set is arbitrary.

Given that we know the underlying structure of the data *a priori*, we can bake this information into the architecture of our deep learning model and effectively provide it with an *inductive bias* that facilitates generalization to unseen instances of the same structure, e.g., combinations of sets with a different number of elements not seen during training.

Here, we introduce the *multi-set convolutional network* (MSCN) model. Our model architecture is inspired by recent work on *Deep Sets* [218], a neural network module for operating on sets. A Deep Sets module (sometimes referred to as *set convolution*) rests on the observation that any function $f(S)$ on a set S that is permutation invariant to the elements in S can be decomposed into the form $\rho[\sum_{x \in S} \phi(x)]$ with appropriately chosen functions ρ and ϕ. For a more

SELECT COUNT(*) FROM title t, movie_companies mc WHERE t.id = mc.movie_id AND t.production_year > 2010 AND mc.company_id = 5
Table set { [0 1 0 1 ... 0], [0 0 1 0 ... 1] } Join set { [0 0 1 0] } Predicate set { [1 0 0 0 0 1 0 0 0.72], [0 0 0 1 0 0 1 0 0.14] }
table id samples join id column id value operator id

Figure 37: Query featurization as sets of feature vectors.

formal discussion and proof of this property, we refer to Zaheer et al. [218]. We choose simple fully-connected multi-layer neural networks (MLPs) to parameterize the functions ρ and ϕ and rely on their function approximation properties [45] to learn flexible mappings $f(S)$ for arbitrary sets S. Since we apply a learnable mapping for each set element individually (with shared parameters), which is similar to the concept of a 1×1 convolution, often used in CNNs for image classification [193], we call our model convolutional.

Our query representation consists of a collection of *multiple* sets, which motivates the following choice for our MSCN model architecture: for every set S, we learn a set-specific, per-element neural network $\text{MLP}_S(v_s)$, i.e., applied on every feature vector v_s for every element $s \in S$ individually[1]. The final representation w_S for this set is then given by the average[2] over the individual transformed representations of its elements, i.e., $w_S = 1/|S| \sum_{s \in S} \text{MLP}_S(v_s)$. We choose an average (instead of, e.g., a simple sum) to ease generalization to different numbers of elements in the set S, as otherwise the overall magnitude of the signal would vary depending on the number of elements in S. In practice, we implement a vectorized version of our model that operates on mini-batches of data. As the number of set elements in each data sample in a mini-batch can vary, we pad all samples with zero-valued feature vectors that act as dummy set elements so that all samples within a mini-batch have the same number of set elements. We mask out dummy set elements in the averaging operation, so that only the original set elements contribute to the average.

Finally, we merge the individual set representations by concatenation and subsequently pass them through a final output MLP:
$w_{\text{out}} = \text{MLP}_{\text{out}}([w_{S_1}, w_{S_2}, \ldots, w_{S_N}])$, where N is the total number of sets and $[\cdot, \cdot]$ denotes vector concatenation. Note that this representation includes the special case where each set representation w_S is transformed by a subsequent individual output function (as required by the original theorem in [218]). One could alternatively process each w_S individually first and only later merge and pass through another MLP. We decided to merge both steps into a single computation for computational efficiency.

1 An alternative approach here would be to combine the feature vectors before feeding them into the MLP. For example, if there are multiple tables, each of them represented by a unique one-hot vector, we could compute the logical disjunction of these one-hot vectors and feed that into the model. Note that this approach does not work if we want to associate individual one-hot vectors with additional information such as the number of qualifying base table samples.

2 Note that an average of one-hot vectors uniquely identifies the combination of one-hot vectors, e.g., which individual tables are present in the query.

Unless otherwise noted, all MLP modules are two-layer fully-connected neural networks with $\text{ReLU}(x) = \max(0, x)$ activation functions. For the output MLP, we use a $\text{sigmoid}(x) = 1/(1 + \exp(-x))$ activation function for the last layer instead and only output a scalar, so that $w_{\text{out}} \in [0, 1]$. We use ReLU activation functions for hidden layers as they show strong empirical performance and are fast to evaluate. All other representation vectors w_T, w_J, w_P, and hidden layer activations of the MLPs are chosen to be vectors of dimension d, where d is a hyperparameter, optimized on a separate validation set via grid search.

We normalize the target cardinalities c_{target} as follows: we first take the logarithm to more evenly distribute target values, and then normalize to the interval $[0, 1]$ using the minimum and maximum value after logarithmization obtained from the training set.[3] The normalization is invertible, so we can recover the unnormalized cardinality from the prediction $w_{\text{out}} \in [0, 1]$ of our model.

We train our model to minimize the mean *q-error* [147] q ($q \geqslant 1$). The q-error is the factor between an estimate and the true cardinality (or vice versa). We further explored using mean-squared error and geometric mean q-error as objectives (cf. Section 4.3.8). We make use of the Adam [98] optimizer for training.

4.2.3 Generating Training Data

One key challenge of all learning-based algorithms is the "cold start problem", i.e., how to train the model before having concrete information about the query workload. Our approach is to obtain an initial training corpus by generating random queries based on schema information and drawing literals from actual values in the database.

A training sample consists of table identifiers, join predicates, base table predicates, and the true cardinality of the query result. To avoid a combinatorial explosion, we only generate queries with up to two joins and let the model generalize to more joins. Our query generator first uniformly draws the number of joins $|J_q|$ $(0 \leqslant |J_q| \leqslant 2)$ and then uniformly selects a table that is referenced by at least one table. For $|J_q| > 0$, it then uniformly selects a new table that can join with the current set of tables (initially only one), adds the corresponding join edge to the query and (overall) repeats this process $|J_q|$ times. For each base table t in the query, it then uniformly draws the number of predicates $|P^t_q|$ $(0 \leqslant |P^t_q| \leqslant \texttt{num non-key columns})$. For each predicate, it uniformly draws the predicate type (=, <, or >) and selects a literal (an actual value) from the corresponding column. We configured our query generator

3 Note that this approach requires complete re-training when data changes (iff the minimum and maximum values have changed). Alternatively, one could set a high limit for the maximum value.

to only generate *unique* queries. We then execute these queries to obtain their true result cardinalities, while skipping queries with empty results. Using this process, we obtain the initial training set for our model.

4.2.4 Enriching the Training Data

A key idea of our approach is to enrich the training data with information about *materialized* base table samples. For each table in a query, we evaluate the corresponding predicates on a materialized sample and annotate the query with the *number* of qualifying samples s ($0 \leqslant s \leqslant 1000$ for 1000 materialized samples) for this table. We perform the same steps for an (unseen) test query at estimation time allowing the ML model to utilize this knowledge.

We even take this idea one step further and annotate each table in a query with the *positions* of the qualifying samples represented as bitmaps (referred to as *sample bitmaps*). As we show in Section 4.3, adding this feature has a positive impact on our join estimates since the ML model can now learn what it means if a certain sample qualifies (e.g., there might be some samples that usually have many join partners). In other words, the model can learn to use the patterns in the bitmaps to predict output cardinalities.

4.2.5 Training and Inference

Building our model involves three steps: (i) generate random (uniformly distributed) queries using schema and data information, (ii) execute these queries to annotate them with their true cardinalities and information about qualifying materialized base table samples, and (iii) feed this training data into an ML model. All of these steps are performed on an immutable snapshot of the database.

To predict the cardinality of a query, the query first needs to be transformed into its feature representation (cf. Section 4.2.1). Inference itself involves a certain number of matrix multiplications, and (optionally) querying materialized base table samples (cf. Section 4.2.4). Training the model with more query samples does not increase the prediction time. In that respect, the inference speed is largely independent from the quality of the predictions. This is in contrast to purely sampling-based approaches that can only increase the quality of their predictions by querying more samples.

4.3 EVALUATION

We evaluate our approach using the IMDb dataset which contains many correlations and therefore proves to be very challenging for cardinality estima-

Table 14: Distribution of joins.

number of joins	0	1	2	3	4	overall
synthetic	1636	1407	1957	0	0	5000
scale	100	100	100	100	100	500
JOB-light	0	3	32	23	12	70

tors [124]. The dataset captures more than 2.5 M movie titles produced over 133 years by 234,997 different companies with over 4 M actors.

We use three different query workloads [122]: (i) a *synthetic* workload generated by the same query generator as our training data (using a different random seed) with 5,000 *unique* queries containing both (conjunctive) equality and range predicates on non-key columns with zero to two joins, (ii) another synthetic workload *scale* with 500 queries designed to show how the model generalizes to more joins, and (iii) *JOB-light*, a workload derived from the Join Order Benchmark (JOB) [124] containing 70 of the original 113 queries. In contrast to JOB, JOB-light does not contain any predicates on strings nor disjunctions and only contains queries with one to four joins. Most queries in JOB-light have equality predicates on dimension table attributes. The only range predicate is on `production_year`. Table 14 shows the distribution of queries with respect to the number of joins in the three query workloads. The non-uniform distribution in the synthetic workload is caused by our elimination of duplicate queries.

As competitors we use PostgreSQL version 10.3, Random Sampling (RS), and Index-Based Join Sampling (IBJS) [127]. RS executes base table predicates on materialized samples to estimate base table cardinalities and assumes independence for estimating joins. If there are no qualifying samples for a conjunctive predicate, it tries to evaluate the conjuncts individually and eventually falls back to using the number of distinct values (of the column with the most selective conjunct) to estimate the selectivity. IBJS represents the state-of-the-art for estimating joins and probes qualifying base table samples against existing index structures. Our IBJS implementation uses the same fallback mechanism as RS.

We train and test our model on an Amazon Web Services (AWS) ml.p2.xlarge instance using the PyTorch framework [172] and use CUDA. We use 100,000 random queries with zero to two joins and 1,000 materialized samples as training data (cf. Section 4.2.3). We split the training data into 90% training and 10% validation samples. To obtain true cardinalities for our training data, we use HyPer [96].

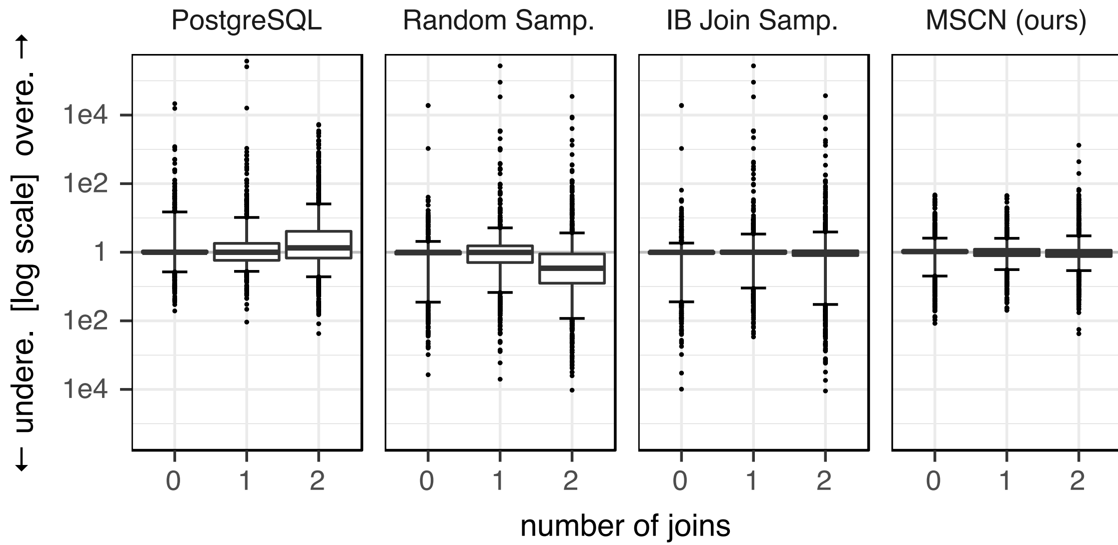


Figure 38: Estimation errors on the synthetic workload. The box boundaries are at the 25th/75th percentiles and the horizontal "whisker" lines mark the 95th percentiles.

Table 15: Estimation errors on the synthetic workload.

	median	90th	95th	99th	max	mean
PostgreSQL	1.69	9.57	23.9	465	373901	154
Random Samp.	1.89	19.2	53.4	587	272501	125
IB Join Samp.	**1.09**	9.93	33.2	295	272514	118
MSCN (ours)	1.18	**3.32**	**6.84**	**30.51**	**1322**	**2.89**

4.3.1 Estimation Quality

Figure 38 shows the q-error of MSCN compared to our competitors. While PostgreSQL's errors are more skewed towards the positive spectrum, RS tends to underestimate joins, which stems from the fact that it assumes independence. IBJS performs extremely well in the median and 75th percentile but (like RS) suffers from empty base table samples. MSCN is competitive with IBJS in the median while being significantly more robust. Considering that IBJS is using much more data—in the form of large primary and foreign key indexes—in contrast to the very small state MSCN is using (less than 3 MiB), MSCN captures (join-crossing) correlations reasonably well and does not suffer as much from 0-tuple situations (cf. Section 4.3.2). To provide more details, we also show the median, percentiles, maximum, and mean q-errors in Table 15. While IBJS provides the best median estimates, MSCN outperforms the competitors by up to two orders of magnitude at the end of the distribution.

Table 16: Estimation errors of 376 base table queries with empty samples in the synthetic workload.

	median	90th	95th	99th	max	mean
PostgreSQL	4.78	62.8	107	1141	21522	133
Random Samp.	9.13	80.1	173	993	19009	147
MSCN	**2.94**	**13.6**	**28.4**	**56.9**	**119**	**6.89**

4.3.2 0-Tuple Situations

Purely sampling-based approaches suffer from empty base table samples (0-tuple situations) which can occur under selective predicates. While this situation can be mitigated using, e.g., more samples or employing more sophisticated—yet still sampling-based—techniques (e.g., [149]), it remains inherently difficult to address by these techniques. In this experiment, we show that deep learning, and MSCN in particular, can handle such situations fairly well.

In fact, 376 (22%) of the 1636 base table queries in the synthetic workload have empty samples (using MSCN's random seed). We will use this subset of queries to illustrate how MSCN deals with situations where it *cannot* build upon (runtime) sampling information (i.e., all sample bitmaps only contain zeros). We also include Random Sampling (which uses the same random seed—i.e., the same set of materialized samples as MSCN) and PostgreSQL in this experiment.

The results, shown in Table 16, demonstrate that MSCN addresses the weak spot of purely sampling-based techniques and therefore would complement them well.

Recall that Random Sampling extrapolates the output cardinality based on the number of qualifying samples (zero in this case). Thus, it cannot simply extrapolate from this number and has to fall back to an educated guess—in our RS implementation either using the product of selectivities of individual conjuncts or using the number of distinct values of the column with the most selective predicate. Independent of the concrete implementation of this fallback, it remains an educated guess. MSCN, in contrast, can use the signal of individual query features (in this case the specific table and predicate features) to provide a more precise estimate.

4.3.3 Removing Model Features

Next, we highlight the contributions of individual model features to the prediction quality (cf. Figure 39). MSCN (no samples) is the model without any (runtime) sampling features, MSCN (#samples) represents the model with one

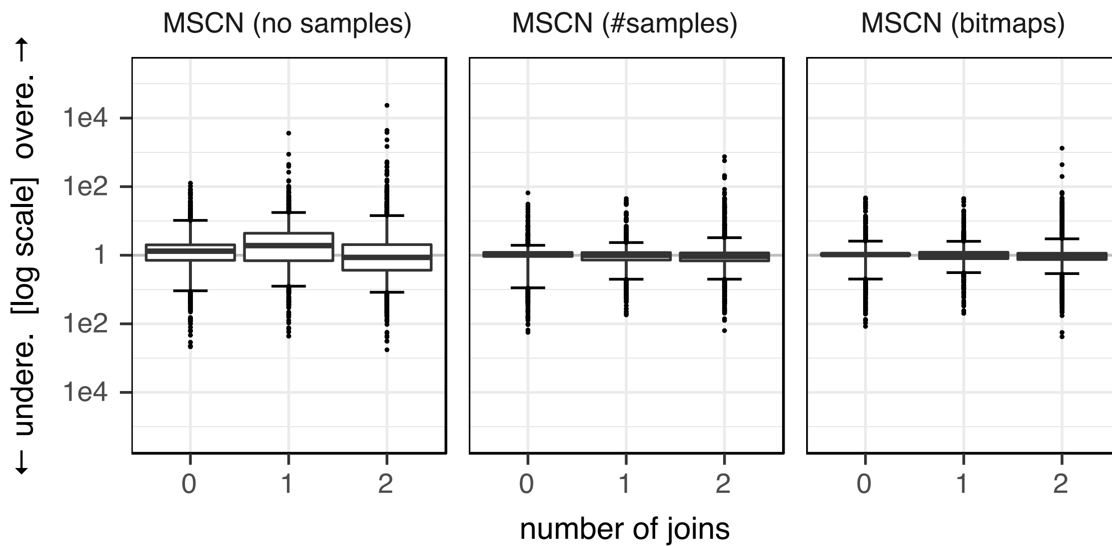


Figure 39: Estimation errors on the synthetic workload with different model variants.

cardinality (i.e., the number of qualifying samples) per base table, and MSCN (bitmaps) denotes the full model with one bitmap per base table.

MSCN (no samples) produces reasonable estimates with an overall 95th percentile q-error of 25.3, purely relying on (inexpensive to obtain) query features. Adding sample cardinality information to the model improves both base table and join estimates. The 95th percentile q-errors of base table, one join, and two join estimates reduce by 1.72×, 3.60×, and 3.61×, respectively. Replacing cardinalities with bitmaps further improves these numbers by 1.47×, 1.35×, and 1.04×. This shows that the model can use the information embedded in the sample bitmaps to provide better estimates.

4.3.4 Generalizing to More Joins

To estimate a larger query, one can of course break the query down into smaller sub queries, estimate them individually using the model, and combine their selectivities. However, this means that we would need to assume independence between two sub queries which is known to deliver poor estimates with real-world datasets such as IMDb (cf. join estimates of Random Sampling in Section 4.3).

The question that we want to answer in this experiment is how MSCN can generalize to queries with more joins than it was trained on. For this purpose, we use the *scale* workload with 500 queries with zero to four joins (100 queries each). Recall that we trained the model only with queries that have between zero and two joins. Thus, this experiment shows how the model can estimate queries with three and four joins *without* having seen such queries during training (cf. Figure 40). From two to three joins, the 95th percentile q-error increases

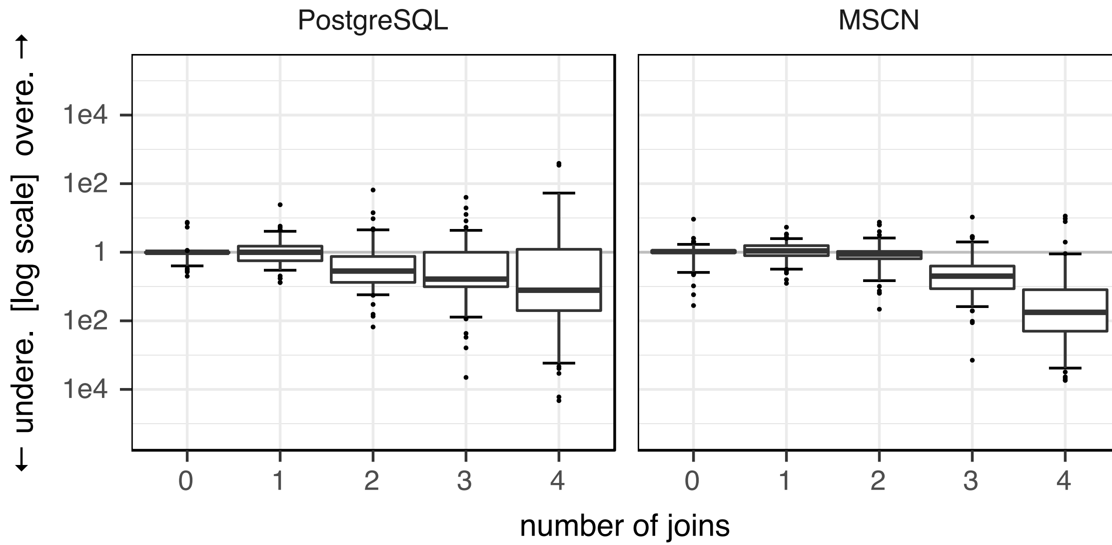


Figure 40: Estimation errors on the scale workload showing how MSCN generalizes to queries with more joins.

Table 17: Estimation errors on the JOB-light workload.

	median	90th	95th	99th	max	mean
PostgreSQL	7.93	164	1104	2912	3477	174
Random Samp.	11.5	198	4073	22748	23992	1046
IB Join Samp.	**1.59**	150	3198	14309	15775	590
MSCN	3.82	**78.4**	**362**	**927**	**1110**	**57.9**

from 7.66 to 38.6. To give a point of reference, PostgreSQL has a 95th percentile q-error of 78.0 for the same queries. And finally, with four joins, MSCN's 95th percentile q-error increases further to 2,397 (PostgreSQL: 4,077).

Note that 58 out of the 500 queries in this workload exceed the maximum cardinality seen during training. 12 of these queries have three joins and another 46 have four joins. When excluding these outliers, the 95th percentile q-errors for three and four joins decrease to 23.8 and 175, respectively.

4.3.5 JOB-light

To show how MSCN generalizes to a workload that was not generated by our query generator, we use JOB-light.

Table 17 shows the estimation errors. Recall that most queries in JOB-light have equality predicates on dimension table attributes. Considering that MSCN was trained with a uniform distribution between =, <, and > predicates, it performs reasonably well. Also, JOB-light contains many queries with a closed range predicate on `production_year`, while the training data only contains

open range predicates. Note that JOB-light also includes five queries that exceed the maximum cardinality that MSCN was trained on. Without these queries, the 95th percentile q-error is 115.

In summary, this experiment shows that MSCN can generalize to workloads with distributions different from the training data.

4.3.6 Hyperparameter Tuning

We tuned the hyperparameters of our model, including the number of epochs (the number of passes over the training set), the batch size (the size of a mini-batch), the number of hidden units, and the learning rate. More hidden units means larger model sizes and increased training and prediction costs with the upside of allowing the model to capture more data, while learning rate and batch size both influence convergence behavior during training.

We varied the number of epochs (100, 200), the batch size (64, 128, 256, 512, 1024, 2048), the number of hidden units (64, 128, 256, 512, 1024, 2048), and fixed the learning rate to 0.001, resulting in 72 different configurations. For each configuration, we trained three models[4] using 90,000 samples and evaluated their performance on the validation set consisting of 10,000 samples. On average over the three runs, the configuration with 100 epochs, a batch size of 1024 samples, and 256 hidden units performed best on the validation data. Across many settings, we observed that 100 epochs perform better than 200 epochs. This is an effect of overfitting: the model captures the noise in the training data such that it negatively impacts its prediction quality. Overall, we found that our model performs well across a wide variety of settings. In fact, the mean q-error only varied by 1% within the best 10 configurations and by 21% between the best and the worst configuration. We also experimented with different learning rates (0.001, 0.005, 0.0001) and found 0.001 to perform best. We thus use 100 epochs, a batch size of 1024, 256 hidden units, and a learning rate of 0.001 as our default configuration.

4.3.7 Model Costs

Next, we analyze the training, inference, and space costs of MSCN with our default hyperparameters. Figure 41 shows how the validation set error (the mean q-error of all queries in the validation set) decreases with more epochs. The model requires fewer than 75 passes (over the 90,000 training queries) to converge to a mean q-error of around 3 on the 10,000 validation queries. An

4 Note that the weights of the neural network are initialized using a different random seed in each training run. To provide reasonably stable numbers, we tested each configuration three times.

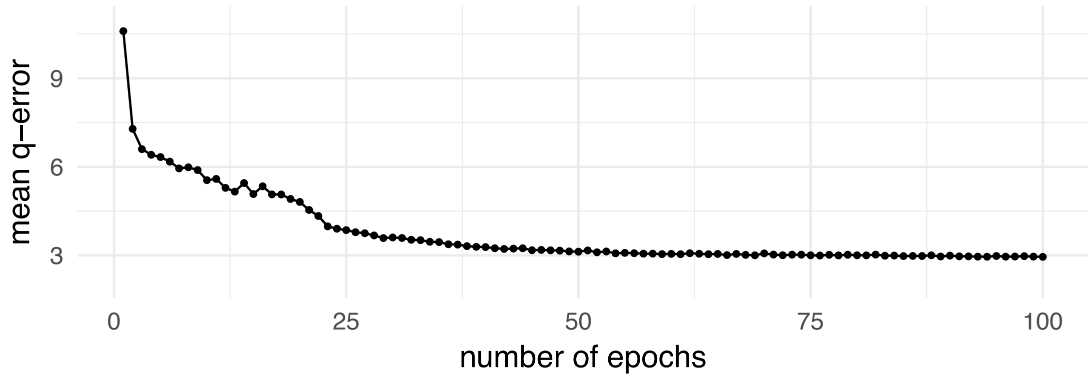


Figure 41: Convergence of the mean q-error on the validation set with the number of epochs.

average training run with 100 epochs (measured across three runs) takes almost 39 minutes.

The prediction time of our model is in the order of a few microseconds per query when batching hundreds of queries. Querying the model with a single query may take much longer due to the overhead introduced by the PyTorch framework. In theory (neglecting the PyTorch overhead), a prediction using a deep learning model (as stated earlier) is dominated by matrix multiplications which can be accelerated using modern GPUs. We thus expect performance-tuned implementations of our model to achieve very low prediction latencies. Since we incorporate sampling information, the end-to-end prediction time will be in the same order of magnitude as that of (per-table) sampling techniques.

The size of our model (when serialized to disk) is 1.6 MiB, 1.6 MiB, and 2.6 MiB for MSCN (no samples), MSCN (#samples), and MSCN (bitmaps), respectively.

4.3.8 Optimization Metrics

Besides optimizing the mean q-error, we also explored using mean-squared error and geometric mean q-error as optimization goals. Mean-squared error would optimize the *squared* differences between the predicted and true cardinalities. Since we are more interested in minimizing the *factor* between the predicted and the true cardinalities (q-error) and use this metric for our evaluation, optimizing the q-error directly yielded better results. Optimizing the geometric mean of the q-error makes the model put less emphasis on heavy outliers (that would lead to large errors). While this approach looked promising at first, it turned out to be not as reliable as optimizing the mean q-error.

4.4 FILTERED GROUP-BY QUERIES

In this section, we discuss another application of ML-based cardinality estimation model, which is predicting the number of unique values in a column or in a combination of columns (i.e., estimating the result size of a group-by operator). This is another hard problem where current approaches achieve undesirable results and where machine learning seems promising.

4.4.1 Problem

The problem is to estimate the number of groups in the output of a group-by query on a base table, such as the following query on the Internet Movie Database (IMDb):

```
SELECT kind_id, phonetic_code, COUNT(*)
FROM title
GROUP BY kind_id, phonetic_code
```

Even for read-only OLAP workloads, this can be a difficult task. While it is trivial to store the number of distinct values of individual columns, it is challenging for multiple columns: That is, because the distinct value counts of individual columns or combinations of columns cannot easily be combined, and consequently, a system would have to maintain an exponential number of such counts to support arbitrary group-by expressions [67].

To work around multi-column statistics, systems like PostgreSQL use sampling to estimate the cardinality of such queries: They take a uniform sample from the relation, and extrapolate statistics obtained on this sample to the whole relation. However, as Charikar et al. proved in their seminal paper [35], it is fundamentally impossible to obtain good estimates from reasonably-sized samples, mainly due to possibly skewed data distributions. In other words, we would have to sample most of the relation in order to reliably achieve high estimation accuracy. As this is clearly not feasible, systems that employ sampling often have no choice but to use wildly inaccurate estimates for query optimization [128].

Freitag et al. thus proposed a hybrid approach [67] that combines single-column statistics with a sampling-based estimator to produce highly accurate multi-column estimates. However, while this approach achieves state-of-the-art estimates and can even deal with updates, we will show in this work that it breaks down when there are selection predicates (filters) on the base table:

```
SELECT kind_id, phonetic_code, COUNT(*)
FROM title
WHERE production_year=2010
GROUP BY kind_id, phonetic_code
```

We will call queries like this *filtered group-by queries* in the following. To accurately predict such queries, the hybrid approach depends on precise estimates of the single-column cardinalities (i.e., `kind_id` and `phonetic_code`) *after* the selection predicate has been applied. Since statistics such as histograms cannot account for selections on dimensions that are not part of the statistic (`production_year` in this example), we must estimate these cardinalities purely based on information obtained from the sample. As outlined above, this will produce provably poor single-column estimates [35], which in turn severely deteriorate the overall accuracy of the estimator. In addition, as all sampling-based estimators, the hybrid approach suffers from 0-tuple situations where no samples remain after the predicate has been applied, leaving it with nothing but an "educated" guess about the number of groups.

Also, even if we would know the total number of groups (without a selection), selectivity estimation alone would not help. We need to estimate the number of *groups* in the result, whereas selectivity estimation can only accurately predict the number of *tuples* [124]. The presence of a selection has an non-trivial effect on the number of groups. For example, constraining `production_year` to 2010 as in the above query results in more than one third of the total number of groups, while setting it to 1900 only yields about 1% of the groups.

Long story short, despite a large volume of previous research activity on the subject [35, 79], traditional approaches continue to struggle with producing accurate cardinality estimates on queries such as filtered group-by queries.

4.4.2 Applications

While our approach only supports the class of filtered group-by queries, its predictions can of course also be used to estimate (sub trees of) larger queries. Another important application is hash table sizing for hash-based aggregations. Also, in distributed query processing, where it is expensive to reoptimize at query runtime (as advocated by [166]), accurately estimating such queries upfront is of high value. For example, it allows the optimizer to decide between a local and a shuffle-based aggregation depending on whether there are few or many unique values, respectively.

4.4.3 Adapting Our Model

In the following, we describe how we adapt our model to estimate filtered group-by queries. From now on, we refer to a trained MSCN model (cf. Section 4.2) and the corresponding base table samples as a *Deep Sketch*. Put differently, a Deep Sketch is a wrapper for a (serialized) neural network and a set of materialized samples.

While MSCN was primarily developed to estimate join queries [100], it can also be applied to the filtered group-by estimation task: The output cardinality of a join query is independent of the concrete query plan—e.g., both $(A \bowtie B) \bowtie C$ and $A \bowtie (B \bowtie C)$ can be represented as a set $\{A, B, C\}$. Similarly, the cardinality of a group-by query does not depend on the concrete permutation of the group-by columns in the SQL string. Thus, we can apply the same set-based model here.

Our original MSCN model represents three sets (tables, joins, and predicates). Here, we use the original join module as a group-by module to estimate filtered group-by queries on base tables. Essentially, instead of encoding identifiers of join predicates, we now encode identifiers of group-by columns.

Similar to the original query featurization (cf. Section 4.2.1), we represent a query $q \in Q$ as a collection (T_q, G_q, P_q) of a set of tables $T_q \subset T$, a set of group-by columns $G_q \subset G$ and a set of (conjunctive) predicates $P_q \subset P$ participating in the specific query q. The only difference here is that we have replaced the original join set with a group-by set, since now we focus on filtered group-by queries on base tables and do not consider joins. We later outline in Section 4.6.6 how to also address joins. T, G, and P describe the entire query space in terms of available tables, group-by columns, and predicates, respectively. Note that here T_q always only consists of a single table.

Figure 42 illustrates the (new) model that operates on the query representation (T_q, G_q, P_q) as follows:

$$
\begin{aligned}
\text{Table module:} \quad & w_T = \frac{1}{|T_q|} \textstyle\sum_{t \in T_q} \text{MLP}_T(v_t) \\
\text{Group-by module:} \quad & w_G = \frac{1}{|G_q|} \textstyle\sum_{g \in G_q} \text{MLP}_G(v_g) \\
\text{Predicate module:} \quad & w_P = \frac{1}{|P_q|} \textstyle\sum_{p \in P_q} \text{MLP}_P(v_p) \\
\text{Merge \& predict:} \quad & w_{out} = \text{MLP}_{out}([w_T, w_G, w_P])
\end{aligned}
$$

We again use the mean q-error [147] q ($q \geqslant 1$) as optimization metric. Similar to our initial approach, we train the model with uniformly distributed queries, but this time only on a single table. The query generator is parametrized with a set of available group-by and selection columns as well as predicate types ($=$, $<$, and $>$). In addition, we specify limits on the number of group-by and selection columns. All decisions are again performed uniformly at random (e.g., how many and which group-by columns a specific query contains). We again execute all training queries against a set of materialized sample tuples to obtain sample bitmaps (cf. Section 4.2.4) and against the full database table to obtain their true cardinalities.

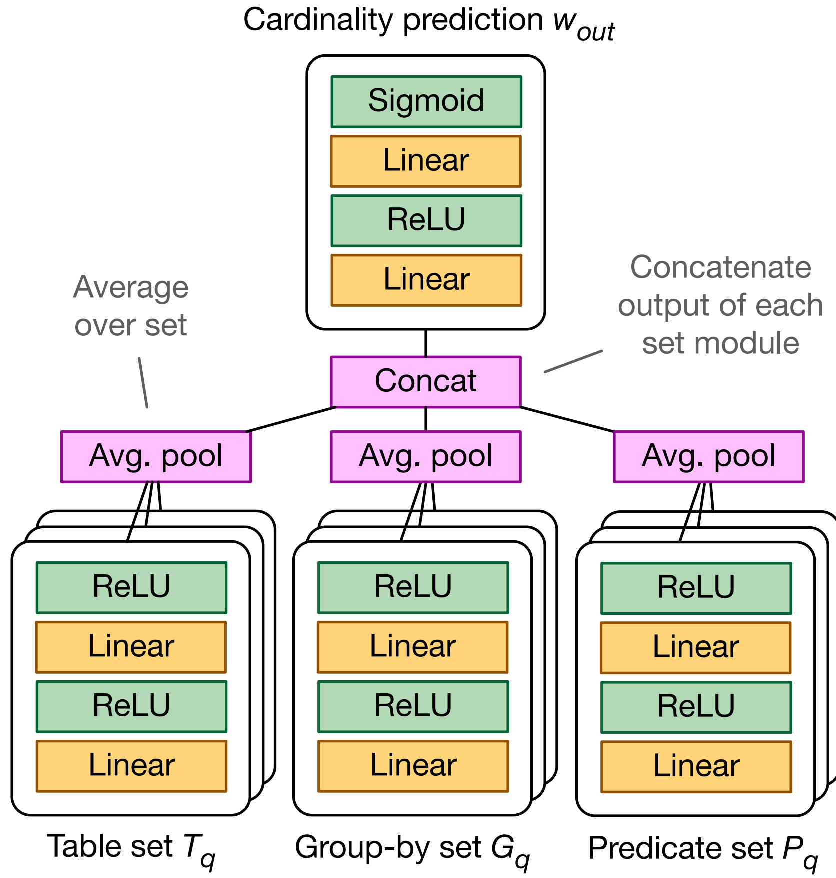


Figure 42: Architecture of the adapted MSCN model. Tables, group-by columns, and predicates are represented as separate MLP modules that provide input to a final output network that predicts query cardinalities.

4.4.4 Results

We now show that our approach can effectively capture correlations between group-by and selection columns. Also, we demonstrate that it significantly outperforms the estimators of PostgreSQL and HyPer [96] as well as the state-of-the-art multi-column estimator (hybrid approach) introduced earlier. Like the state size of the hybrid approach, the size of our ML model only grows linearly with the number of columns while it can accurately predict the effect of selections.

Dataset and Query Workload

We again use the real-world IMDb dataset, which is challenging for cardinality estimators [124] due to data skew and correlations. This time, we focus our evaluation on the `title` table.

Table 18: Cardinalities (distinct value counts) of columns used in our workload. Checkmarks indicate whether columns can appear in filter and/or group-by clauses.

column	cardinality	filter	group by
kind_id	6		✓
phonetic_code	23259		✓
episode_nr	14907	✓	✓
production_year	133	✓	✓

In the following, we describe our query workload including the search space (space of possible queries) that it is drawn from.

We generate a workload that operates on four columns (cf. Table 18). The construction of a query works as follows (all decisions are performed uniformly at random): We generate either one or two filter predicates (selections) on the filter columns, i.e., either one filter (on `episode_nr` *or* on `production_year`) or two filters (on `episode_nr` *and* on `production_year`). Each filter can be an equality (=), a less than (<), or a greater than (>) predicate. Filter literals are drawn from the respective columns in the database (e.g., 2010 for a filter on `production_year`). In fact, we pick values from random (uniformly distributed) row offsets. Next, we generate the group-by clause. We generate either one or two group-by columns, chosen among all four columns (without replacement).

Thus, given the distinct value counts shown in Table 18, there are 5,993,013 possible selections and 10 possible group-by clauses, resulting in 59,930,130 possible *unique* queries. For our query workload, we use 500 queries from within that search space.

Deep Sketch

We use a Deep Sketch (DS) that is trained with uniformly distributed queries within the above query space. The training set is *disjoint* from the query workload. By default, we use 10,000 training queries and 1,000 materialized samples (i.e., each query is annotated with 1,000 bits).

We use the MSCN model with the following hyperparameters: 100 epochs, batch size of 1024, 256 hidden units, and a learning rate of 0.001. Training the model with 10,000 queries takes around two minutes on a GeForce GTX 1050 Ti GPU using the PyTorch framework [172] with CUDA. Querying the model with 500 annotated queries takes around two microseconds per query when running the model on the GPU. Note that this does not include the time for executing the query against the table sample. When serialized to disk, the model itself consumes 2.5 MiB (and another 16 KiB are occupied by the table sample). Note that its size increases linearly with the number of columns: Since

columns are encoded as unique one-hot vectors, we require one additional 256-dimensional vector (# of hidden units) for each extra column.

Competitors

We compare against PostgreSQL version 10.3 and HyPer as well as the state-of-the-art multi-column estimator SCBC (hybrid approach) [67]. At its core, SCBC employs an improved sampling-based estimator derived from the estimators proposed by Charikar et al. [35]. For our experiments, this part of SCBC uses the same sample than the Deep Sketch. As outlined previously, a sampling-only approach will have provably poor accuracy in some cases. For this reason, SCBC additionally maintains the distinct value counts in the individual columns. In order to obtain a multi-column estimate, SCBC computes lower and upper bounds on the number of groups based on these single-column counts, which are then used to constrain the output of the sampling-based estimator.

Without any selections, this approach gives excellent estimates due to the accurate single-column estimates. In order to extend SCBC to filtered group-by queries, we can simply execute the query on the sample, and run the sampling-based part of SCBC afterwards. However, we cannot predict the impact of selections on the number of distinct values in the individual columns accurately. This prevents SCBC from computing a reliable lower bound on the number of groups after selections. For the purposes of this work, we only compute an upper bound on the number of groups based on the single-column cardinalities *before* any selections. Note that for very selective queries, this bound will be much too loose, severely limiting the performance of SCBC.

During our preliminary evaluation, we have tried to compute single-column estimates after selections. However, without any additional information, this boils down to a sample-based distinct value count estimation, which gives poor results [35]. As SCBC heavily relies on accurate single-column estimates, such an approach cannot improve the accuracy of SCBC on filtered group-by queries.

Estimation Quality

Figure 43 and Table 19 show the q-errors of the different approaches. While PostgreSQL and HyPer achieve reasonable median q-errors, they both have heavy outliers. SCBC has a lower median q-error than PostgreSQL and HyPer, but also suffers from large mispredictions. These mispredictions arise mainly due to the presence of selections in the workload. As outlined above, selections prevent SCBC from computing tight bounds on the result cardinality, and there are many cases in which it has to rely purely on the potentially inaccurate sampling-based estimator. DS achieves the best median estimation quality and is significantly more robust than its competitors. In fact, the estimates

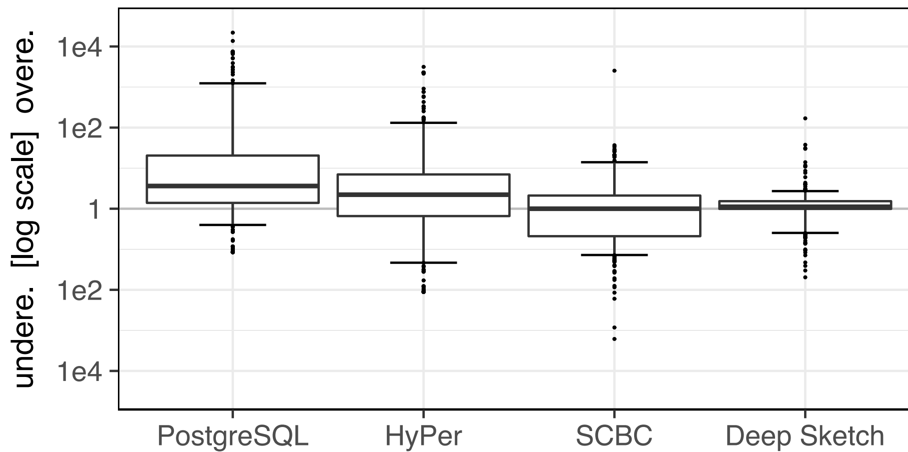


Figure 43: Estimation errors on the query workload. The box boundaries are at the 25th/75th percentiles and the horizontal "whisker" lines mark the 95th percentiles.

Table 19: Estimation errors on the query workload.

	median	90th	95th	99th	max	mean
PostgreSQL	4.35	318	1236	7431	21934	351
HyPer	6.00	111	131	2275	3144	74.1
SCBC	3.21	14.0	21.0	86.8	2528	16.6
Deep Sketch	**1.31**	**3.00**	**5.12**	**30.0**	**169**	**2.58**

produced by DS are within a factor of 3 of the true cardinalities on average. Note that DS only observed 10,000 of the almost 60 M possible queries during training.

Effect of Number of Training Queries

We now analyze the performance of DS with fewer training queries. Figure 44 shows the q-errors of DS with a varying number of training queries. With only 100 queries, DS already achieves a median q-error of 4.33 which is competitive with the other approaches. However, as one would expect with such a small training dataset, it produces large outliers. Starting with 1,000 training queries, DS surpasses HyPer in the 95th percentile. With 5,000 queries, it eventually outperforms SCBC in all metrics.

Effect of Selections

We now use another (manually constructed) query workload to highlight the effect of selections. Like in our opening example, we have an equality

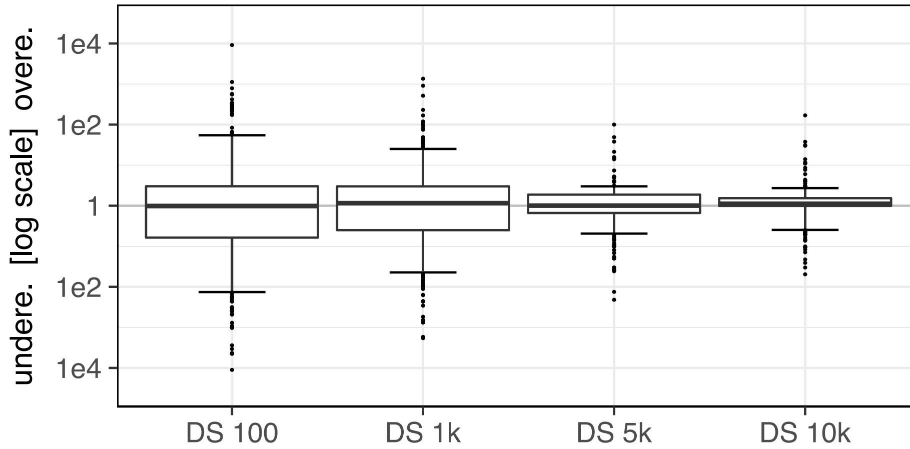


Figure 44: Estimation errors on the query workload with an increasing number of training queries.

predicate on `production_year` (and vary the literal from 1880 to 2020) and use the `kind_id` and `phonetic_code` columns in the group-by clause.

The Deep Sketch is again trained using the same 10,000 queries as above.

Here, we also include a simple combinatorial estimation approach (which we call *baseline*) that has access to the total number of groups G and the true number of qualifying tuples t. Note that in reality we do not know the exact values of these variables. This approach assumes a uniform distribution of values among groups and works as follows: Assume a relation contains G groups and $N = G \times n$ tuples with n being the number of tuples per group. A query selects t tuples randomly without replacement, i.e., we disregard a possible correlation between selection and group-by columns. Then the expected number of groups in the result set is:

$$G \times \left(1 - \frac{\binom{N-n}{t}}{\binom{N}{t}}\right)$$

Figure 45 shows the results with the four approaches in different columns. The x-axis denotes the production year while we plot the cardinalities on the y-axis. The green line shows the true cardinalities, while the red line denotes the estimated cardinalities. In addition, the *dotted* red line shows the estimates of the baseline approach.

We first only group by the low cardinality column `kind_id` (first row in Figure 45). As it turns out (green line), the number of different movie kinds increases over time. For example, in 1931 the first *tv series* came out. Before that year, all titles were of kind *movie*, *tv movie*, or *episode*. Our baseline (dotted red line) overestimates the number of distinct movie kinds early on. It essentially assumes that the number of distinct movie kinds directly correlates with the number of movie titles per year.

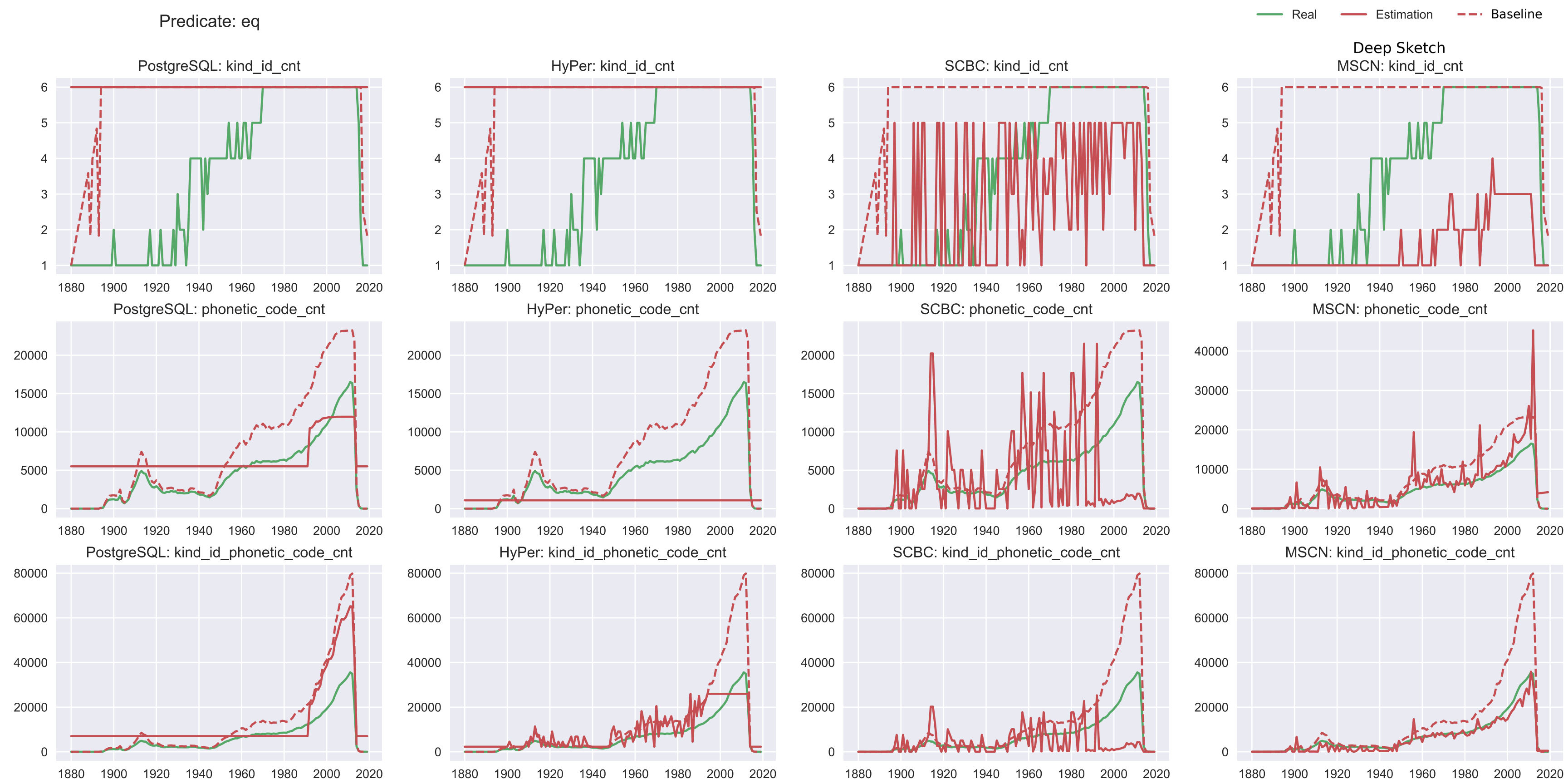


Figure 45: Cardinality estimates (y-axis) of PostgreSQL, HyPer, SCBC, and Deep Sketches. Selection on `production_year` (x-axis) and group-by on `kind_id` and/or `phonetic_code`.

In fact, once the number of yearly movie titles increases (which it actually does), the baseline estimates more distinct movie kinds. The reason for its overestimation is that it assumes that the qualifying movie titles t are uniformly distributed among movie kinds. In reality, however, this distribution is non-uniform and highly correlated with the year. Independent of the year, PostgreSQL and HyPer always estimate 6 distinct movie kinds, which is the total number of distinct values (cf. Table 18). SCBC's estimates fluctuate between 1 and 5. This result further illustrates the importance of tight bounds on the output cardinality for SCBC. The predicates in this workload are generally quite selective, i.e., the number of groups in the output is small. However, SCBC only has access to the single-column cardinalities *before* these selections, which provide no useful upper bound on the number of groups in the output. Thus, SCBC almost exclusively relies on its sampling-based estimator, which leads to the observed severe overestimations.

Moreover, SCBC also suffers heavily from 0-tuple situations, which explains the observed frequent underestimations. Our query workload contains 132 different equality predicates on `production_year`, whereas the sample used by SCBC only contains 100 distinct values of `production_year`. Thus, there are 32 queries in which the sample will contain no tuples after applying the selection, and SCBC can only guess the output cardinality. Since no lower bound on the number of groups in the output is available, SCBC can also not correct this guess reasonably. As Figure 45 illustrates, DS does not suffer from this problem.

In order to confirm these considerations, we ran the same workload but gave SCBC access to the true column cardinalities *after* selections. As expected, this enabled SCBC to predict the result cardinalities with high accuracy.

DS captures the increase in movie kinds but still underestimates it. The reason is that certain kinds are rare and are thus less likely to be captured during training.

Next, we use the high cardinality column `phonetic_code` in the group-by clause (second row in Figure 45). Again, the number of distinct values essentially increases over time (green line), with a sharp rise after 1980. Our baseline (dotted red line) produces reasonable estimates up to the year 1940 before starting to overestimate the cardinality in recent years. The reason is again that it assumes a direct correlation between the number of yearly movie titles and the number of distinct phonetic codes. PostgreSQL's and HyPer's estimates are again hardly unaffected by the selection. While PostgreSQL catches the increase in distinct values in recent years, HyPer heavily underestimates all of these queries. SCBC's estimates again fluctuate due to the same reasons outlined above. DS estimates closely approximate the true cardinalities with a few exceptions.

Finally, we group by both `kind_id` and `phonetic_code` (third row in Figure 45). The trend (green line) is similar than for the queries that only group by

phonetic_code (second row), but interestingly the estimations of PostgreSQL, HyPer, and SCBC differ: PostgreSQL estimates the spike in recent years to be larger this time, while HyPer's estimates are now influenced by the selection. SCBC's estimates still fluctuate but are now much closer to the true cardinalities, except in recent years where its sampling-based estimator underestimates the number of groups. DS again produces highly accurate estimates with minimal outliers.

In summary, all of the traditional approaches have issues with this workload. DS, in contrast, captures the correlations between the selection on production_year and the group-by columns, leading to more accurate estimates.

4.5 OPTIMIZER INTEGRATION

In this section, we outline how learned cardinalities could be leveraged by existing query optimizers. Particularly, we describe a prototype that demonstrates the end-to-end training and cardinality estimation process. Here, we focus on the original MSCN model (cf. Section 4.2), without the filtered group-by adaptation (cf. Section 4.4).

4.5.1 Overview

As stated earlier, a Deep Sketch is essentially a wrapper for a trained MSCN model and a set of materialized samples. Our prototype allows to define such sketches, monitor the training process, and run ad-hoc queries against trained sketches. While the focus here is *not* to show the effect of better cardinality estimates on the quality of resulting query plans—which is orthogonal to having better estimates in the first place, we demonstrate that the estimates produced by Deep Sketches are superior to estimates of traditional optimizers (including the cardinality estimators of HyPer [96] and PostgreSQL [171]) and often close to the ground truth. The estimates produced by Deep Sketches can directly be leveraged by existing, sophisticated join enumeration algorithms and cost models. This is a more gradual approach than the one taken by machine learning (ML)-based end-to-end query optimizers [140, 141].

Our demonstration allows users to define Deep Sketches on the TPC-H and Internet Movie Database (IMDb) datasets. To create a Deep Sketch, users select a subset of tables and define a few parameters such as the number of training queries. Users can then monitor the training progress, including the execution of training queries and the training of the deep learning model. Once the model has been trained, users can issue ad-hoc queries against the resulting Deep Sketch. Our user interface makes it easy to create such queries graphically. Users can optionally specify a placeholder for a certain column to define

a query template. For example, a movie producer might be interested in the popularity of a certain keyword over time:

```
SELECT COUNT(*)
FROM title t, movie_keyword mk, keyword k
WHERE mk.movie_id=t.id AND mk.keyword_id=k.id
AND k.keyword='artificial-intelligence'
AND t.production_year=?
```

Another example—from the domain of sports performance analysis—is the occurrences of certain actions (e.g., number of passes in soccer) over the duration of a game. A placeholder has a similar effect as a group-by operation, except that it does not operate on all distinct values of the group-by column but instead only on the values present in the column sample that comes with the sketch. In other words, we instantiate the query template with values (literals) from the column sample. Besides this being an interesting feature for data analysts, it serves the purpose of visualizing the robustness of our deep learning approach to cardinality estimation.[5] The result of a query template can be displayed as a bar or as a line plot with one data point per template instance. Using overlays, we show the difference to the cardinality estimates produced by HyPer and PostgreSQL as well as to the true cardinalities—obtained by executing the queries with HyPer.

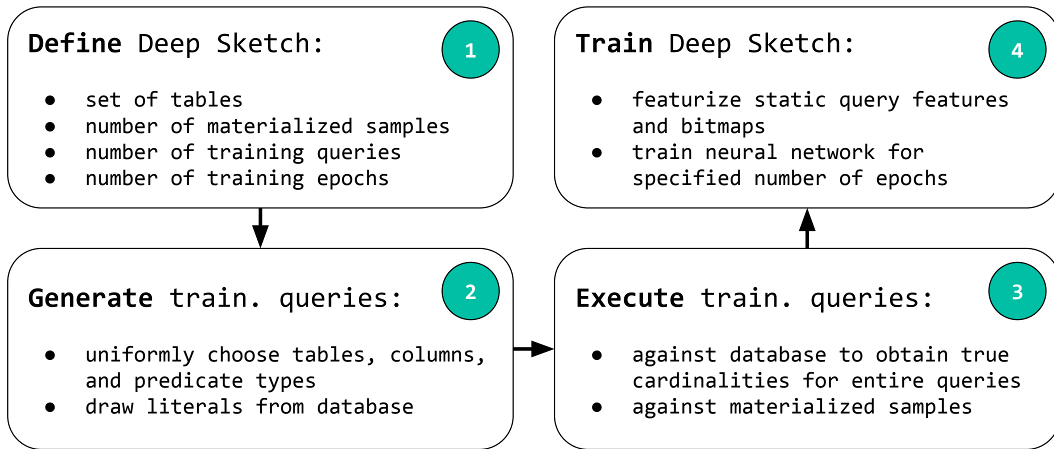
4.5.2 Training and Query Flow

In our demonstration, users can experience the end-to-end process of defining, training, and using trained Deep Sketches to estimate the result sizes of ad-hoc SQL queries. We support the TPC-H and IMDb datasets.

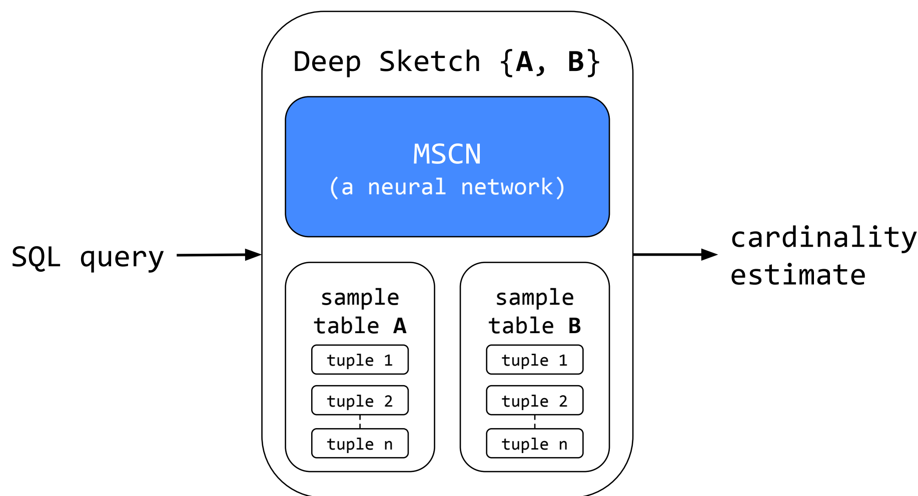
Figure 46a shows the four steps involved to create a new sketch. First (1), users need to select a subset of tables from either schema and define a few parameters, including the number of materialized base table samples, the number of training queries, and the number of training epochs. Next (2), we generate uniformly distributed training queries on the specified tables in our backend, and (3) execute them with HyPer to obtain true cardinalities and to extract bitmaps indicating qualifying samples. To accelerate this process during our demonstration, we plan to execute the training queries (in parallel) on multiple HyPer instances. Finally (4), we featurize the training queries and train the MSCN model for the specified number of epochs.

To give a point of reference on the training costs, training the model with 90,000 queries over 100 epochs takes almost 39 minutes on an Amazon Web Services (AWS) ml.p2.xlarge instance using the PyTorch framework [172] with

5 Note that the deep learning model is not necessarily trained with literals present in the column sample. In fact, it can happen (and is even likely for columns with many distinct values) that a literal from the column sample has never been seen by the model.



(a) Creation of a sketch.



(b) Result size estimation with a sketch.

Figure 46: Creation and usage of a Deep Sketch. Depending on the number of training queries, training can be expensive. However, once a sketch is trained, it allows for an efficient result size estimation of SQL queries.

CUDA. Since this number is too high for an interactive user experience, we address this problem in three ways.

First, we allow users to control the number of training queries and epochs. For a small number of tables, 10,000 queries will already be sufficient to achieve good results. Note that the training time decreases linearly with fewer epochs. From our experience, 25 epochs are usually enough to achieve a reasonable mean q-error on a separate validation set. Second, we offer pre-built (high quality) models that can be queried right away. Third, we allow users to train new models *while* querying existing ones.

Figure 46b illustrates a Deep Sketch on two tables A and B. The interface of a sketch is very simple, it consumes a SQL query and returns a cardinality estimate. First, the query's selection predicates are evaluated on the respec-

Figure 47: Web interface for Deep Sketches.

tive table sample to obtain bitmaps indicating qualifying samples, before the query and its bitmaps are featurized and passed through MSCN to produce an estimate.

4.5.3 Web Interface

Figure 47 shows our web interface for Deep Sketches. On the left, we allow users to specify SQL queries, and on the right, we display query results (estimated and true cardinalities). On the top (`SHOW SKETCHES`), users can select existing and create new sketches as described above. To query a sketch, users do not need enter SQL directly (the SQL string shown in the web interface is only displayed for information purposes). Instead, we provide them with a simple graphical query interface. By clicking on a table, it is added to the query. When a user selects multiple tables, we automatically add the corresponding join predicates to the query (based on the single PK/FK relationships that exist between tables).

Users can also define selections on base tables by clicking the respective columns in the schema. We support both equality and range predicates. In addition, we allow users to specify a placeholder for a certain column. Since our Deep Sketch implementation can only estimate single queries, we automatically instantiate such query templates and—in the background—execute each instance separately against the sketch. To create such an instance, we draw a value from the column sample that is part of the sketch. Optionally, users can select a *function* to be applied to these values. For example, for columns with many distinct values—such as `Date` columns, users may want to "group" the results by year (e.g., `EXTRACT(YEAR FROM date)`). To serve such queries, we

generate multiple range queries (one for each year found in the sample) to be issued against the sketch. We also support grouping the output into equally sized buckets based on the minimum and maximum values from the sample.

When a user hits the `EXECUTE` button, we issue the query against HyPer to compute its true cardinality as well as against the Deep Sketch and the cardinality estimators of HyPer and PostgreSQL to obtain estimates. The query results are displayed with different overlays as they arrive. On the x-axis we denote values from the placeholder column and on the y-axis we plot the estimated and true cardinalities. We support both bar and line charts and allow users to hide the results of individual systems.

4.5.4 Open Challenges

Our demonstration allows users to experience the end-to-end training and querying process of a learned cardinality model. However, to automate these processes in a query optimizer, clearly more research is needed. Decision points—that we currently outsource to our users—include for which schema parts we should build such sketches and when to refresh them.

4.6 DISCUSSION

We have shown that our model can beat state-of-the-art approaches for cardinality estimation. It does a good job in addressing 0-tuple situations and in capturing join-crossing correlations, especially when combined with runtime sampling. To make it suitable for general-purpose cardinality estimation, it can be extended into multiple dimensions, including complex predicates, uncertainty estimation, and updatability. In the following, we will discuss these and sketch possible solutions.

4.6.1 Generalization

MSCN can to some extent generalize to queries with more joins than seen during training (cf. Section 4.3.4). Nevertheless, generalizing to queries that are not in the vicinity of the training data remains challenging.

Of course, our model can be trained with queries from an actual workload or their structures. In practice, we could replace any literals in user queries with placeholders to be filled with actual values from the database. This would allow us to focus on the relevant joins and predicates.

4.6.2 Adaptive Training

To improve training quality, we could *adaptively* generate training samples: based on the error distribution of queries in the validation set, we would generate new training samples that shine more light on difficult parts of the schema.

4.6.3 Strings

A simple addition to our current implementation are equality predicates on strings. To support these, we could hash the string literals to a (small) integer domain. Then an equality predicate on a string is essentially the same as an equality predicate on an ID column where the model also needs to process a non-linear input signal.

4.6.4 Complex Predicates

Currently, our model can only estimate queries with predicate types that it has seen during training. Complex predicates, such as LIKE or disjunctions, are not yet supported since we currently do not represent them in the model. An idea to allow for any complex predicate would be to purely rely on the sample bitmaps in such cases. Note that this would make our model vulnerable to 0-tuple situations. To mitigate that problem, we could featurize information from histograms. Also, the distribution of bitmap patterns might vary significantly from simple predicates observed at training time, to more complicated predicates at test time, which can make generalization challenging.

4.6.5 More Bitmaps

At the moment, we use a single bitmap indicating the qualifying samples per base table. To increase the likelihood for qualifying samples, we could additionally use one bitmap per predicate. For example, for a query with two conjunctive base table predicates, we would have one bitmap for each predicate, and another bitmap representing the conjunction. In a column store that evaluates one column at a time, we can obtain this information almost for free. We have already shown that MSCN can use the information embedded in the bitmaps to make better predictions. We expect that it would benefit from the patterns in these additional bitmaps.

This approach should also help MSCN with estimating queries with arbitrary (complex) predicates where it needs to rely on information from the (many) bitmaps. Of course, this approach does not work in 0-tuple situations, or more specifically in situations where none of the (predicate) bitmaps indicates any qualifying samples.

4.6.6 Filtered Group-By Estimates

Next, we discuss ways to improve the results of our model for estimating filtered group-by queries (cf. Section 4.4).

More Runtime Features

Currently, we only featurize bitmaps indicating table samples that have survived the selection. Since we need to consult the sample anyways, we could also derive and encode further statistics, such as the number of distinct values of individual group-by columns in the sample *before* and *after* selection, normalized based on the number of distinct values in the entire table. Similarly, we could compute and encode the number of groups in the sample *before* and *after* selection (again normalized based on table statistics).

Combining DS and SCBC

While we have shown that DS only needs to observe 0.02% of the query space to produce highly accurate estimates for our query workload, this might not always be the case. To prune the search space and thus reduce the number of required training queries, we could combine DS with SCBC. We have previously shown that SCBC produces accurate group-by estimates *without* selections when it has access to precise estimates of the number of distinct values of individual columns [67]. Under selections, SCBC does not have such estimates and as Charikar et al. proved [35], also cannot compute precise estimates based on a sample. To provide SCBC with precise estimates of single-column distinct value counts *after* selection, we could train a DS specifically for this task. By focusing on single group-by columns, we would effectively prune the search space and would need fewer training queries. The downside is that a combined DS/SCBC approach would still suffer from 0-tuple situations, i.e., when the sampling-based estimator of SCBC has no qualifying tuples to start with. DS on its own does not have this problem and can still rely on static query features (i.e., group-by columns and predicates) in such cases [100].

Supporting Joins

In the current filtered group-by model, we have replaced the original join module with a group-by module. Instead, we could add a forth set module to capture group-by and join queries. While this is a rather straightforward extension, this would increase the query space and thus require a larger training dataset.

4.6.7 Uncertainty Estimation

An open question is when to actually trust the model and rely on its predictions. One approach is to use strict constraints for generating the training data and enforce them at runtime, i.e., only use the model when all constraints hold (i.e., only PK/FK joins, only equality predicates on certain columns). A more appealing approach would be to implement uncertainty estimation into the model. However, for a model like ours, this is a non-trivial task and still an area of active research. There are some recent methods [74, 97, 117] that we plan to investigate in future work.

4.6.8 Updates

Throughout this work, we have assumed an immutable (read-only) database. To handle data and schema changes, we can either completely re-train the model or we can apply some modifications to our model that allow for incremental training.

Complete re-training comes with considerable compute costs (for re-executing queries to obtain up-to-date cardinalities and for re-training the model) but would allow us to use a different data encoding. For example, we could use larger one-hot vectors to accommodate for new tables and we could re-normalize values in case of new minimum or maximum values. Queries (training samples) of which we know to still have the same cardinality (e.g., since there has not been any update to the respective data range) would of course not need to re-executed.

In contrast, incremental training (as implied by its name) would not require us to re-train with the original set of samples. Instead, we could re-use the model state and only apply new samples. One challenge with incremental training is to accommodate changes in the data encoding, including one-hot encodings and the normalization of values. To recall, there are two types of values that we normalize: literals in predicates (actual column values) and output cardinalities (labels). For both types, setting a high limit on the maximum value seems most appropriate. The main challenge, however, is to address *catastrophic forgetting*, which is an effect that can be observed with neural networks when data distribution shifts over time. The network would overfit to the most recent data and forget what it has learned in the past. Addressing this problem is an area of active research with some recent proposals [107].

4.7 RELATED WORK

Deep learning has been applied to query optimization by three recent papers [140, 161, 112] that formulate join ordering as a *reinforcement learning* problem and use ML to find *query plans*. This work, in contrast, applies *supervised learning* to solve *cardinality estimation* in isolation. This focus is motivated by the fact that modern join enumeration algorithms can find the optimal join order for queries with dozens of relations [155]. Cardinality estimation, on the other hand, has been called the "Achilles heel" of query optimization [135] and causes most of its performance issues [124].

4.7.1 ML-Based Approaches

Twenty years ago the first approaches to use neural networks for cardinality estimation where published for UDF predicates [116]. Also, regression-based models have been used before for cardinality estimation [5]. A semi-automatic alternative for explicit machine learning was presented in [137], where the feature space is partitioned using decision trees and for each split a different regression model was learned. These early approaches did not use deep learning nor included features derived from statistics, such as our sample-based bitmaps, which encode exactly which sample tuples were selected (and we therefore believe to be good starting points for learning correlations). The same holds for approaches that used machine learning to predict overall resource consumption: running time, memory footprint, I/O, network traffic [131, 69], although these models did include course-grained features (the estimated cardinality) based on statistics into the features. Liu et al. [133] used modern ML for cardinality estimation, but did not focus on joins, which are the key estimation challenge [124].

4.7.2 Sampling

Our approach builds on sampling-based estimation by including cardinalities or bitmaps derived from samples into the training signal. Most sampling proposals create per-table samples/sketches and try to combine them intelligently in joins [60, 207, 201, 41]. While these approaches work well for single-table queries, they do not capture join-crossing correlations and are vulnerable to the 0-tuple problem (cf. Section 4.3.2). Recent work by Müller et al. [149] aims to reduce the 0-tuple problem for conjunctive predicates (albeit at high computational cost), but still cannot capture the basic case of a single predicate giving zero results. Our reasonably good estimates in 0-tuple situations make MSCN improve over sampling, including even the idea of estimation on

materialized join samples (join synopses [169]), which still would not handle 0-tuple situations.

4.7.3 Group-By Estimates

Given its importance for query optimization [124], there is also a plethora of approaches for estimating the number of distinct values in a column. Traditionally, systems maintain some sort of statistics on base tables, and attempt to derive cardinality estimates from these statistics [124, 146]. For this purpose, sampling is one of the most versatile approaches, as it offers attractive performance and naturally deals with selections and multi-column estimates. However, as Charikar et al. proved, any exclusively sampling-based approach must have poor accuracy on some input data [35]. A large body of work thus attempts to improve sampling-based estimation with auxiliary information [73, 53, 127, 121, 219]. This includes a state-of-the-art hybrid estimator [67], which, as opposed to previous approaches, can produce accurate multi-column estimates with minimal overhead but struggles in the presence of selections.

Current systems frequently assume that the individual attributes are independent for multi-column estimation, and use ad-hoc heuristics to estimate the impact of selections [124]. However, such heuristics frequently do not reflect real-world data accurately, resulting in large estimation errors [214]. Multi-column statistics, for example multi-dimensional histograms [183], or wavelets [43], promise more accurate cardinality estimates, but as opposed to Deep Sketches, their space consumption is non-linear in the number of attributes.

While our approach uses supervised ML, there is also work on using unsupervised ML for cardinality estimation. However, none of these approaches have been adapted to group-by queries. The closest work is [80] in which the authors propose to use an autoregressive model to learn a conditional probability distribution. However, their approach only supports equality predicates and does not address group-by queries. Another recent proposal is to use deep generative models to capture the joint data distributions of multiple attributes and to generate new sample tuples following that distribution [195]. The idea then is to run actual queries on these representative samples. Besides its application in approximate query processing, one could use that approach to estimate cardinalities of filtered group-by queries. However, since the sample tuples are generated from the joint distribution over all columns, selective queries may require a high number of samples to achieve good approximation performance.

4.8 CONCLUSIONS

We have introduced a new approach to cardinality estimation based on MSCN, a new deep learning model. We have trained MSCN with generated queries, uniformly distributed within a constrained search space. We have shown that it can learn (join-crossing) correlations and that it addresses the weak spot of sampling-based techniques, which is when no samples qualify. We have extended our approach to filtered group-by queries and have demonstrated its high accuracy. We have presented a prototype that showcases the query optimizer integration. Our model is a first step towards reliable ML-based cardinality estimation and can be extended into multiple dimensions, including complex predicates, uncertainty estimation, and updatability.

5 FUTURE WORK

In this thesis, we have made multiple contributions to the research area of analytical database systems. In Chapter 2, we have studied how main-memory database systems (MMDBs) can be extended to match the performance and usability of streaming systems. While we have addressed performance issues regarding networking and scalability, we have not considered the usability aspects. SQL is arguably the standard for data analysis today and in the foreseeable future. However, streaming support in SQL is still limited and not available in many database systems. While Apache Flink and other streaming systems allow for event time semantics (e.g., for windowed aggregations), SQL simply does not. In a recent proposal [21], Begoli et al. describe a set of extensions to SQL to better support streaming use cases. Once part of the SQL standard, such a consolidation between traditional SQL (on tables) and streaming SQL (on streams) can be a powerful tool to close the gap between MMDBs and stream data processing even further.

In Chapter 3, we have introduced a novel radix tree-based polygon index that accelerates point-polygon joins in main memory. We have demonstrated the high performance of this approach, which is even competitive with state-of-the-art GPU implementations. However, we currently use a sparse tree node representation that is not very space efficient and can lead to multiple GiBs of memory consumption when indexing many polygons. Recent work on succinct trie data structures [223] suggests that a space-efficient tree representation can still achieve comparable lookup performance to pointer-based data structures like ours. Along these lines, hybrid data structures [222, 7] that combine the benefits of succinct and pointer-based data structures seem to be a promising way forward. Likewise, learned indexes [110] promise both space efficiency and high lookup performance. The idea is to approximate the cumulative distribution function (CDF) of the data with a model to predict the estimated position of the lookup key in a sorted array. Given that our cell identifiers are one-dimensional, learned indexes would be directly applicable to this problem. However, whether such a model can be competitive with an optimized radix tree for *prefix lookups* remains to be seen. To save on valuable main memory, the use of non-volatile memory (NVM) for storing lower levels of the tree could be a good strategy as well.

In Chapter 4, we have proposed a new deep learning approach to cardinality estimation that captures correlations between columns, even across tables. Our model incorporates both query and runtime features (sample bitmaps). While we have shown that our approach achieves state-of-the-art results on

some cardinality estimation tasks, its applicability to real-world systems is limited. At the time of writing this thesis, there have been a number of follow-ups (most notably [211, 205, 160, 84, 81, 57, 192]) to our initial proposal addressing some of its weaknesses. Despite our integration of runtime features, predicting queries that are not in the vicinity of the training data remains challenging. Similarly, low selectivity queries with no or only a few qualifying samples pose severe problems to our model, especially when considering that similar queries have likely not been encountered during training. In that regard, the unsupervised learning method Naru [211] has recently re-defined the state of the art, outperforming supervised approaches and sampling by a large margin. While Naru is limited to equality and range predicates, it could well be combined with supervised (query-driven) methods or sampling to extend its scope. Hilprecht et al. [84] also propose to learn from data, not from queries, to avoid expensive training data collection. Furthermore, they also cover joins and allow for incremental model updates to reflect data changes. Woltmann et al. suggest to pre-compute joins between tables of interest and build *local* deep learning models on each of these subsets. Along the same lines, Wu et al. [206] propose to focus the training on specific query templates to reduce the search space. Sun et al. [192] improve on our model by supporting complex (string) predicates using a preprocessing phase. To better support larger joins, Hayek et al. [81] build on query containment rate prediction with a query pool and show promising improvements over our method. Finally, all of the approaches above use deep network architectures. Despite advances in hardware, these models remain compute intense, both during training and inference. To make ML-based cardinality estimation practical, Dutt et al. argue to use *lightweight* models such as gradient boosted trees [40] and show competitive results. A recent survey on learning-based cardinality estimation by Ortiz et al. [160] confirms that simple models are sufficient in many cases. But more importantly, this study also finds that improved estimates from these models lead to significantly better query plans. The question that remains is whether we should learn cardinalities, cost models [192], or even entire query optimizers [138]. In our experience, starting "simple" (with cardinalities) is difficult enough. Thus, we believe that we should first "solve" the cardinality estimation in isolation before increasing complexity. A path forward would be to focus on learning cardinalities that make a difference to plans produced by the optimizer.

BIBLIOGRAPHY

In compliance with § 6 Abs. 6 Satz 3 Promotionsordnung der Technischen Universität München, publications by the author of this thesis are marked with an asterisk (*).

[1] Danial Aghajarian and Sushil K. Prasad. "A Spatial Join Algorithm Based on a Non-uniform Grid Technique over GPGPU". In: *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2017, Redondo Beach, CA, USA, November 7-10, 2017*, 56:1–56:4.

[2] Divyakant Agrawal, Amr El Abbadi, Ambuj K. Singh, and Tolga Yurek. "Efficient View Maintenance at Data Warehouses". In: *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*. Ed. by Joan Peckham. ACM Press, pp. 417–427.

[3] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. "Weaving Relations for Cache Performance". In: *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pp. 169–180.

[4] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz. "Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce". In: *PVLDB* 6.11 (2013), pp. 1009–1020.

[5] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. "Learning-based Query Performance Modeling and Prediction". In: *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pp. 390–401.

[6] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. "Automatic Database Management System Tuning Through Large-scale Machine Learning". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pp. 1009–1024.

[7] * Christoph Anneser, Andreas Kipf, Harald Lang, Thomas Neumann, and Alfons Kemper. "The Case for Hybrid Succinct Data Structures". In: *Proceedings of the 23nd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pp. 391–394.

[8] *Apache Calcite*. `https://calcite.apache.org/`.

[9] *Apache Storm Trident State*. `http://storm.apache.org/releases/current/Trident-state.html`.

[10] Joy Arulraj, Andrew Pavlo, and Subramanya Dulloor. "Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pp. 707–722.

[11] Joy Arulraj, Ran Xian, Lin Ma, and Andrew Pavlo. "Predictive Indexing". In: *CoRR* abs/1901.07064 (2019).

[12] Artin Avanes. *Automatic Clustering, Materialized Views & Automatic Maintenance in Snowflake.* 2018. URL: `https://www.snowflake.com/blog/automatic-clustering-materialized-views-automatic-maintenance-in-snowflake/` (visited on 12/03/2019).

[13] AWS. *Amazon Redshift announces support for spatial data*. 2019. URL: `https://aws.amazon.com/about-aws/whats-new/2019/11/amazon-redshift-announces-support-spatial-data/` (visited on 11/30/2019).

[14] AWS. *Amazon Redshift introduces Automatic Table Sort, an automated alternative to Vacuum Sort*. 2019. URL: `https://aws.amazon.com/about-aws/whats-new/2019/11/amazon-redshift-introduces-automatic-table-sort-alternative-vacuum-sort/` (visited on 11/30/2019).

[15] *AWS Instance Types*. `https://aws.amazon.com/ec2/instance-types/`.

[16] Leonardo Guerreiro Azevedo, Ralf Hartmut Güting, Rafael Brand Rodrigues, Geraldo Zimbrão, and Jano Moreira de Souza. "Filtering with raster signatures". In: *14th ACM International Symposium on Geographic Information Systems, ACM-GIS 2006, November 10-11, 2006, Arlington, Virginia, USA, Proceedings*, pp. 187–194.

[17] Leonardo Guerreiro Azevedo, Geraldo Zimbrão, and Jano Moreira de Souza. "Approximate Query Processing in Spatial Databases Using Raster Signatures". In: *VIII Brazilian Symposium on Geoinformatics, 19-22 November, Campos do Jordão, São Paulo, Brazil*, pp. 53–72.

[18] Nagender Bandi, Chengyu Sun, Amr El Abbadi, and Divyakant Agrawal. "Hardware Acceleration in Commercial Databases: A Case Study of Spatial Operations". In: *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pp. 1021–1032.

[19] Jeff Barr. *EC2 High Memory Update – New 18 TB and 24 TB Instances*. 2019. URL: `https://aws.amazon.com/blogs/aws/ec2-high-memory-update-new-18-tb-and-24-tb-instances/` (visited on 12/01/2019).

[20] Jeff Barr. *Elastic Network Adapter - High Performance Network Interface for Amazon EC2*. 2016. URL: `https://aws.amazon.com/blogs/aws/elastic-network-adapter-high-performance-network-interface-for-amazon-ec2/` (visited on 10/22/2019).

[21] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. "One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. Ed. by Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska. ACM, pp. 1757–1772.

[22] *BigQuery*. `https://cloud.google.com/bigquery/`.

[23] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. "The End of Slow Networks: It's Time for a Redesign". In: *PVLDB* 9.7 (2016), pp. 528–539.

[24] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (1970), pp. 422–426.

[25] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. "Breaking the memory wall in MonetDB". In: *Commun. ACM* 51.12 (2008), pp. 77–85.

[26] *boost::geometry::index::rtree - 1.60.0*. `https://www.boost.org/doc/libs/1_60_0/libs/geometry/doc/html/geometry/reference/spatial_indexes/boost__geometry__index__rtree.html`.

[27] *NYC Boroughs*. `https://data.cityofnewyork.us/City-Government/Borough-Boundaries/tqmj-j8zm`.

[28] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Scalable Garbage Collection for In-Memory MVCC Systems". In: *PVLDB* 13.2 (2019), pp. 128–141.

[29] Lucas Braun. "Confidentiality and Performance for Cloud Databases". PhD thesis. ETH Zurich, 2017.

[30] Lucas Braun, Thomas Etter, Georgios Gasparis, Martin Kaufmann, Donald Kossmann, Daniel Widmer, Aharon Avitzur, Anthony Iliopoulos, Eliezer Levy, and Ning Liang. "Analytics in Motion: High Performance Event-Processing AND Real-Time Analytics in the Same Database". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pp. 251–264.

[31] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. "Multi-Step Processing of Spatial Joins". In: *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994.* Pp. 197–208.

[32] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *IEEE Data Eng. Bull.* 38.4 (2015), pp. 28–38.

[33] *NYC Census Blocks*. `https://data.cityofnewyork.us/City-Government/2010-Census-Blocks/v2h8-6mxf`.

[34] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. "Trill: A High-Performance Incremental Query Processor for Diverse Analytics". In: *PVLDB* 8.4 (2014), pp. 401–412.

[35] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. "Towards Estimation Error Guarantees for Distinct Values". In: *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, May 15-17, 2000, Dallas, Texas, USA*, pp. 268–279.

[36] Biswapesh Chattopadhyay et al. "Procella: Unifying serving and analytical data at YouTube". In: *PVLDB* 12.12 (2019), pp. 2022–2034.

[37] Surajit Chaudhuri, Gautam Das, and Vivek R. Narasayya. "A Robust, Optimization-Based Approach for Approximate Answering of Aggregate Queries". In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*. Ed. by Sharad Mehrotra and Timos K. Sellis. ACM, pp. 295–306.

[38] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. "Approximate Query Processing: No Silver Bullet". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pp. 511–519.

[39] Harshada Chavan, Rami Alghamdi, and Mohamed F. Mokbel. "Towards a GPU accelerated spatial computing framework". In: *32nd IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2016, Helsinki, Finland, May 16-20, 2016*, pp. 135–142.

[40] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*. Ed. by Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi. ACM, pp. 785–794.

[41] Yu Chen and Ke Yi. "Two-Level Sampling for Join Size Estimation". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pp. 759–774.

[42] Sanket Chintapalli et al. "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, pp. 1789–1792.

[43] Graham Cormode, Minos N. Garofalakis, Peter J. Haas, and Chris Jermaine. "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches". In: *Foundations and Trends in Databases* 4.1-3 (2012), pp. 1–294.

[44] *Google C++ B-Tree*. `https://code.google.com/archive/p/cpp-btree/`.

[45] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *MCSS* 2.4 (1989), pp. 303–314.

[46] Benoît Dageville et al. "The Snowflake Elastic Data Warehouse". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pp. 215–226.

[47] Guilherme Damasio, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Alexandar Mihaylov, Jaroslaw Szlichta, and Calisto Zuzarte. "Guided automated learning for query workload re-optimization". In: *PVLDB* 12.12 (2019), pp. 2010–2021.

[48] *Data Plane Development Kit*. `https://www.dpdk.org/`.

[49] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. "Monkey: Optimal Navigable Key-Value Store". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pp. 79–94.

[50] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pp. 4171–4186.

[51] F van Diggelen and P Enge. "The world's first GPS MOOC and worldwide laboratory using smartphones". In: *Proc. of ION GNSS+*, 361–369.

[52] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. "AI Meets AI: Leveraging Query Executions to Improve Index Recommendations". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pp. 1241–1258.

[53] Bolin Ding, Silu Huang, Surajit Chaudhuri, Kaushik Chakrabarti, and Chi Wang. "Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pp. 679–694.

[54] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. "Optimizing Space Amplification in RocksDB". In: *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*.

[55] Harish Doraiswamy, Eleni Tzirita Zacharatou, Fábio Miranda, Marcos Lage, Anastasia Ailamaki, Cláudio T. Silva, and Juliana Freire. "Interactive Visual Exploration of Spatio-Temporal Urban Data Sets using Urbane". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pp. 1693–1696.

[56] *DriveNow*. https://www.drive-now.com/de/en.

[57] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. "Selectivity Estimation for Range Predicates using Lightweight Models". In: *PVLDB* 12.9 (2019), pp. 1044–1057.

[58] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim M. Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. "Reducing DRAM footprint with NVM in facebook". In: *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, 42:1–42:13.

[59] Ahmed Eldawy. "SpatialHadoop: towards flexible and scalable spatial processing using mapreduce". In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, Utah, USA, June 22, 2014, PhD Symposium*, pp. 46–50.

[60] Cristian Estan and Jeffrey F. Naughton. "End-biased Samples for Join Cardinality Estimation". In: *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, p. 20.

[61] Yi Fang, Marc Friedman, Giri Nair, Michael Rys, and Ana-Elisa Schmid. "Spatial indexing in microsoft SQL server 2008". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pp. 1207–1216.

[62] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. "SAP HANA database: data management for modern business applications". In: *SIGMOD Record* 40.4 (2011), pp. 45–51.

[63] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. "The SAP HANA Database – An Architecture Overview". In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 28–33.

[64] * Philipp Fent, Michael Jungmair, Andreas Kipf, and Thomas Neumann. "START — Self-Tuning Adaptive Radix Tree". In: *36th IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2020, Dallas, TX, USA, April 20-24, 2020*, pp. 147–153.

[65] * Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory". In: *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*.

[66] Samuel Flender. *Data is not the new oil*. 2019. URL: `https://towardsdatascience.com/data-is-not-the-new-oil-bdb31f61bc2d` (visited on 11/27/2019).

[67] Michael J. Freitag and Thomas Neumann. "Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*.

[68] Edward Gan, Jialin Ding, Kai Sheng Tai, Vatsal Sharan, and Peter Bailis. "Moment-Based Quantile Sketches for Efficient High Cardinality Aggregation Queries". In: *PVLDB* 11.11 (2018), pp. 1647–1660.

[69] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. "Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning". In: *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pp. 592–603.

[70] Venkatesh Ganti, Mong-Li Lee, and Raghu Ramakrishnan. "ICICLES: Self-Tuning Samples for Approximate Query Answering". In: *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*. Ed. by Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang. Morgan Kaufmann, pp. 176–187.

[71] Irene Gargantini. "An Effective Way to Represent Quadtrees". In: *Commun. ACM* 25.12 (1982), pp. 905–910.

[72] Gartner. *Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015*. 2015. URL: `https://www.gartner.com/en/newsroom/press-releases/2015-11-10-gartner-says-6-billion-connected-things-will-be-in-use-in-2016-up-30-percent-from-2015` (visited on 10/22/2019).

[73] Phillip B. Gibbons. "Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports". In: *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pp. 541–550.

[74] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. "On Calibration of Modern Neural Networks". In: *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pp. 1321–1330.

[75] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. "Amazon Redshift and the Case for Simpler Data Warehouses". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pp. 1917–1923.

[76] Ashish Gupta et al. "Mesa: Geo-Replicated, Near Real-Time, Scalable Data Warehousing". In: *PVLDB* 7.12 (2014), pp. 1259–1270.

[77] Eric Haines. "Point in polygon strategies". In: *Graphics gems IV* 994 (1994), pp. 24–26.

[78] Theo Härder and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery". In: *ACM Comput. Surv.* 15.4 (1983), pp. 287–317.

[79] Hazar Harmouch and Felix Naumann. "Cardinality Estimation: An Experimental Survey". In: *PVLDB* 11.4 (2017), pp. 499–512.

[80] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. "Multi-Attribute Selectivity Estimation Using Deep Learning". In: *CoRR* abs/1903.09999 (2019).

[81] Rojeh Hayek and Oded Shmueli. "Improved Cardinality Estimation by Learning Queries Containment Rates". In: *CoRR* abs/1908.07723 (2019).

[82] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. "Generalized Search Trees for Database Systems". In: *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland.* Pp. 562–573.

[83] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. "Towards learning a partitioning advisor with deep reinforcement learning". In: *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*, 6:1–6:4.

[84] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. "DeepDB: Learn from Data, not from Queries!" In: *CoRR* abs/1909.00607 (2019).

[85] Clive Humby. *Data is the New Oil*. 2006. URL: `https://ana.blogs.com/maestros/2006/11/data_is_the_new.html` (visited on 11/27/2019).

[86] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. "Database Cracking". In: *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pp. 68–78.

[87] Edwin H. Jacox and Hanan Samet. "Spatial join techniques". In: *ACM Trans. Database Syst.* 32.1 (2007), p. 7.

[88] Shrainik Jain, Jiaqi Yan, Thierry Cruanes, and Bill Howe. "Database-Agnostic Workload Management". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*.

[89] Eunyoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. "mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems". In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pp. 489–502.

[90] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pp. 1–12.

[91] Robert Kallman et al. "H-store: a high-performance, distributed main memory transaction processing system". In: *PVLDB* 1.2 (2008), pp. 1496–1499.

[92] Srikanth Kandula, Kukjin Lee, Surajit Chaudhuri, and Marc Friedman. "Experiences with Approximating Queries in Microsoft's Production Big-Data Clusters". In: *PVLDB* 12.12 (2019), pp. 2131–2142.

[93] Kothuri Venkata Ravi Kanth and Siva Ravada. "Efficient Processing of Large Spatial Queries Using Interior Approximations". In: *Advances in Spatial and Temporal Databases, 7th International Symposium, SSTD 2001, Redondo Beach, CA, USA, July 12-15, 2001, Proceedings*, pp. 404–424.

[94] Kothuri Venkata Ravi Kanth, Siva Ravada, and Daniel Abugov. "Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data". In: *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pp. 546–557.

[95] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. "Benchmarking Distributed Stream Processing Engines". In: *CoRR* abs/1802.08496 (2018).

[96] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots". In: *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pp. 195–206.

[97] Alex Kendall and Yarin Gal. "What Uncertainties Do We Need in Bayesian Deep Learning for Computer Vision?" In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pp. 5574–5584.

[98] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

[99] * Andreas Kipf, Michael Freitag, Dimitri Vorona, Peter Boncz, Thomas Neumann, and Alfons Kemper. "Estimating Filtered Group-By Queries is Hard: Deep Learning to the Rescue". In: *1st International Workshop on Applied AI for Database Systems and Applications* (2019).

[100] * Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. "Learned Cardinalities: Estimating Correlated Joins with Deep Learning". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*.

[101] * Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Christoph Anneser, Eleni Tzirita Zacharatou, Harish Doraiswamy, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Adaptive Main-Memory Indexing for High-Performance Point-Polygon Joins". In: *Proceedings of the 23nd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, pp. 347–358.

[102] * Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Approximate Geospatial Joins with Precision Guarantees". In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pp. 1360–1363.

[103] * Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. "SOSD: A Benchmark for Learned Indexes". In: *NeurIPS Workshop on Machine Learning for Systems* (2019).

[104] * Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. "Analytics on Fast Data: Main-Memory Database Systems versus Modern Streaming Systems". In: *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017.* Pp. 49–60.

[105] * Andreas Kipf, Varun Pandey, Jan Böttcher, Lucas Braun, Thomas Neumann, and Alfons Kemper. "Scalable Analytics on Fast Data". In: *ACM Trans. Database Syst.* 44.1 (2019), 1:1–1:35.

[106] * Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Estimating Cardinalities with Deep Sketches". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* Pp. 1937–1940.

[107] James Kirkpatrick et al. "Overcoming catastrophic forgetting in neural networks". In: *CoRR* abs/1612.00796 (2016).

[108] Allen Klinger. "Patterns and search statistics". In: *Optimizing methods in statistics*. Elsevier, 1971, pp. 303–337.

[109] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. "SageDB: A Learned Database System". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*.

[110] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. "The Case for Learned Index Structures". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pp. 489–504.

[111] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: A distributed messaging system for log processing". In: *Proceedings of the NetDB*, pp. 1–7.

[112] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. "Learning to Optimize Join Queries With Deep Reinforcement Learning". In: *CoRR* abs/1808.03196 (2018).

[113] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, pp. 1106–1114.

[114] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. "Fast Updates on Read-Optimized Databases Using Multi-Core CPUs". In: *PVLDB* 5.1 (2011), pp. 61–72.

[115] Moritz Kulessa, Alejandro Molina, Carsten Binnig, Benjamin Hilprecht, and Kristian Kersting. "Model-based Approximate Query Processing". In: *CoRR* abs/1811.06224 (2018).

[116] M. Seetha Lakshmi and Shaoyu Zhou. "Selectivity Estimation in Extensible Databases - A Neural Network Approach". In: *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pp. 623–627.

[117] Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. "Simple and Scalable Predictive Uncertainty Estimation using Deep Ensembles". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pp. 6402–6413.

[118] * Harald Lang, Andreas Kipf, Linnea Passing, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines". In: *Proceedings of the 14th International Workshop on Data Management on New Hardware, Houston, TX, USA, June 11, 2018*, 5:1–5:8.

[119] Harald Lang, Thomas Neumann, Alfons Kemper, and Peter A. Boncz. "Performance-Optimal Filtering: Bloom overtakes Cuckoo at High-Throughput". In: *PVLDB* 12.5 (2019), pp. 502–515.

[120] * Harald Lang, Linnea Passing, Andreas Kipf, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines". In: *VLDBJ* (2019), pp. 1–18.

[121] Per-Åke Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. "Cardinality estimation using sample views with quality assurance". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pp. 175–186.

[122] *Learned Cardinalities in PyTorch*. https://github.com/andreaskipf/learnedcardinalities.

[123] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age". In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pp. 743–754.

[124] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. "How Good Are Query Optimizers, Really?" In: *PVLDB* 9.3 (2015), pp. 204–215.

[125] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. "LeanStore: In-Memory Data Management beyond Main Memory". In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pp. 185–196.

[126] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The adaptive radix tree: ARTful indexing for main-memory databases". In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pp. 38–49.

[127] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. "Cardinality Estimation Done Right: Index-Based Join Sampling". In: *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*.

[128] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. "Query optimization through the looking glass, and what we found running the Join Order Benchmark". In: *VLDB J.* 27.5 (2018), pp. 643–668.

[129] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. "The ART of practical synchronization". In: *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, 3:1–3:8.

[130] Chao Li, Yi Yang, Min Feng, Srimat T. Chakradhar, and Huiyang Zhou. "Optimizing memory efficiency for deep convolutional neural networks on GPUs". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, pp. 633–644.

[131] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. "Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques". In: *PVLDB* 5.11 (2012), pp. 1555–1566.

[132] Xi Liang, Aaron J. Elmore, and Sanjay Krishnan. "Opportunistic View Materialization with Deep Reinforcement Learning". In: *CoRR* abs/1903-.01363 (2019).

[133] Henry Liu, Mingbin Xu, Ziting Yu, Vincent Corvinelli, and Calisto Zuzarte. "Cardinality estimation using neural networks". In: *Proceedings of 25th Annual International Conference on Computer Science and Software Engineering, CASCON 2015, Markham, Ontario, Canada, 2-4 November, 2015*, pp. 53–59.

[134] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. "On the Design and Scalability of Distributed Shared-Data Databases". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pp. 663–676.

[135] Guy Lohman. *Is Query Optimization a Solved Problem?* 2014. URL: `http://wp.sigmod.org/?p=1075` (visited on 11/14/2019).

[136] Lothar F. Mackert and Guy M. Lohman. "R* Optimizer Validation and Performance Evaluation for Distributed Queries". In: *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pp. 149–159.

[137] Tanu Malik, Randal C. Burns, and Nitesh V. Chawla. "A Black-Box Approach to Query Cardinality Estimation". In: *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pp. 56–67.

[138] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. "Neo: A Learned Query Optimizer". In: *PVLDB* 12.11 (2019), pp. 1705–1718.

[139] Ryan C. Marcus and Olga Papaemmanouil. "Plan-Structured Deep Neural Network Models for Query Performance Prediction". In: *PVLDB* 12.11 (2019), pp. 1733–1746.

[140] Ryan Marcus and Olga Papaemmanouil. "Deep Reinforcement Learning for Join Order Enumeration". In: *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, 3:1–3:4.

[141] Ryan Marcus and Olga Papaemmanouil. "Towards a Hands-Free Query Optimizer through Deep Learning". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*.

[142] Charles Masson, Jee E. Rim, and Homin K. Lee. "DDSketch: A Fast and Fully-Mergeable Quantile Sketch with Relative-Error Guarantees". In: *PVLDB* 12.12 (2019), pp. 2195–2205.

[143] Friedemann Mattern and Christian Floerkemeier. "From the Internet of Computers to the Internet of Things". In: *From Active Data Management to Event-Based Systems and More - Papers in Honor of Alejandro Buchmann on the Occasion of His 60th Birthday*, pp. 242–259.

[144] John Meehan et al. "S-Store: Streaming Meets Transaction Processing". In: *PVLDB* 8.13 (2015), pp. 2134–2145.

[145] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. "Efficient Computation of Frequent and Top-k Elements in Data Streams". In: *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings*, pp. 398–412.

[146] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. "Why go logarithmic if we can go linear?: Towards effective distinct counting of search traffic". In: *EDBT 2008, 11th International Conference on Extending Database Technology, Nantes, France, March 25-29, 2008, Proceedings*, pp. 618–629.

[147] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. "Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors". In: *PVLDB* 2.1 (2009), pp. 982–993.

[148] Tobias Mühlbauer, Wolf Rödiger, Angelika Reiser, Alfons Kemper, and Thomas Neumann. "ScyPer: A Hybrid OLTP&OLAP Distributed Main Memory Database System for Scalable Real-Time Analytics". In: *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 11.-15.3.-2013 in Magdeburg, Germany. Proceedings*, pp. 499–502.

[149] Magnus Müller, Guido Moerkotte, and Oliver Kolb. "Improved Selectivity Estimation by Combining Knowledge from Sampling and Synopses". In: *PVLDB* 11.9 (2018), pp. 1016–1028.

[150] *NYC Neighborhoods*. `https://data.cityofnewyork.us/City-Government/Neighborhood-Tabulation-Areas/cpf4-rkhq`.

[151] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware". In: *PVLDB* 4.9 (2011), pp. 539–550.

[152] Thomas Neumann. *Random Execution Plans*. 2014. URL: `http://databasearchitects.blogspot.com/2014/06/random-execution-plans.html` (visited on 11/28/2019).

[153] Thomas Neumann and Alfons Kemper. "Unnesting Arbitrary Queries". In: *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, pp. 383–402.

[154] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. "Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pp. 677–689.

[155] Thomas Neumann and Bernhard Radke. "Adaptive Optimization of Very Large Join Queries". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pp. 677–692.

[156] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. "Samza: stateful scalable stream processing at LinkedIn". In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1634–1645.

[157] *NVIDIA NVLink*. `https://www.nvidia.com/en-us/data-center/nvlink/`.

[158] *NVIDIA Tesla V100*. `https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf`.

[159] Jack A. Orenstein. "Redundancy in Spatial Databases". In: *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989.* Pp. 295–305.

[160] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. "An Empirical Analysis of Deep Learning for Cardinality Estimation". In: *CoRR* abs/1905.06425 (2019).

[161] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. "Learning State Representations for Query Optimization with Deep Reinforcement Learning". In: *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, 4:1–4:4.

[162] * Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. "How Good Are Modern Spatial Analytics Systems?" In: *PVLDB* 11.11 (2018), pp. 1661–1673.

[163] * Varun Pandey, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper. "High-Performance Geospatial Analytics in HyPerSpace". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pp. 2145–2148.

[164] Andrew Pavlo et al. "Self-Driving Database Management Systems". In: *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*.

[165] Ross Perez. *How Snowpipe Streamlines Your Continuous Data Loading and Your Business*. 2017. URL: `https://www.snowflake.com/blog/snowpipe-serverless-loading-for-streaming-data/` (visited on 11/30/2019).

[166] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. "How I Learned to Stop Worrying and Love Re-optimization". In: *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pp. 1758–1761.

[167] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. "Fast Scans on Key-Value Stores". In: *PVLDB* 10.11 (2017), pp. 1526–1537.

[168] *PipelineDB*. `https://www.pipelinedb.com/`.

[169] Viswanath Poosala and Yannis E. Ioannidis. "Selectivity Estimation Without the Attribute Value Independence Assumption". In: *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pp. 486–495.

[170] *PostGIS: Spatial and Geographic objects for PostgreSQL*. `http://postgis.net/`.

[171] *PostgreSQL*. `http://www.postgresql.org/`.

[172] *PyTorch*. `https://pytorch.org/`.

[173] Mark Raasveldt and Hannes Mühleisen. "Don't Hold My Data Hostage - A Case For Client Protocol Redesign". In: *PVLDB* 10.10 (2017), pp. 1022–1033.

[174] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. "Managing Non-Volatile Memory in Database Systems". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, 1541–1555.

[175] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Persistent Memory I/O Primitives". In: *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN 2019, Amsterdam, The Netherlands, 1 July 2019*, 12:1–12:7.

[176] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. "High-Speed Query Processing over High-Speed Networks". In: *PVLDB* 9.4 (2015), pp. 228–239.

[177] Jörg Roth. "The extended split index to efficiently store and retrieve spatial data with standard databases". In: *Proceedings of the IADIS International Conference Applied Computing 2009, 19-21 November, Rome, Italy, 2 Volumes*, pp. 85–92.

[178] *Google S2 Library*. http://s2geometry.io/.

[179] *S2Geometry Basic Types*. http://s2geometry.io/devguide/basic_types.html.

[180] *Google S2ShapeIndex*. https://s2geometry.io/devguide/s2shapeindex.

[181] Bart Samwel et al. "F1 Query: Declarative Querying at Scale". In: *PVLDB* 11.12 (2018), pp. 1835–1848.

[182] Grigory Sapunov. *Hardware for Deep Learning. Part 3: GPU*. 2018. URL: https://blog.inten.to/hardware-for-deep-learning-part-3-gpu-8906c1644664 (visited on 12/03/2019).

[183] Michael Shekelyan, Anton Dignös, and Johann Gamper. "DigitHist: a Histogram-Based Data Summary with Tight Error Bounds". In: *PVLDB* 10.11 (2017), pp. 1514–1525.

[184] Darius Sidlauskas, Sean Chester, Eleni Tzirita Zacharatou, and Anastasia Ailamaki. "Improving Spatial Data Processing by Clipping Minimum Bounding Boxes". In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*, pp. 425–436.

[185] Gabriel Silva. *Maximize your VM's Performance with Accelerated Networking*. 2018. URL: https://azure.microsoft.com/en-us/blog/maximize-your-vm-s-performance-with-accelerated-networking-now-generally-available-for-both-windows-and-linux/ (visited on 10/22/2019).

[186] *Solarflare OpenOnload*. https://www.openonload.org/.

[187] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. "LEO - DB2's LEarning Optimizer". In: *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pp. 19–28.

[188] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. "The 8 requirements of real-time stream processing". In: *SIGMOD Record* 34.4 (2005), pp. 42–47.

[189] *STX B+-tree*. http://panthema.net/2007/stx-btree/.

[190] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. "Exploring Spatial Datasets with Histograms". In: *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pp. 93–102.

[191] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. "Hardware Acceleration for Spatial Selections and Joins". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pp. 455–466.

[192] Ji Sun and Guoliang Li. "An End-to-End Learning-based Cost Estimator". In: *PVLDB* 13.3 (2019), pp. 307–319.

[193] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. "Rethinking the Inception Architecture for Computer Vision". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, 2818–2826.

[194] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. "LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data". In: *PVLDB* 9.13 (2016), 1565–1568.

[195] Saravanan Thirumuruganathan, Shohedul Hasan, Nick Koudas, and Gautam Das. "Approximate Query Processing using Deep Generative Models". In: *CoRR* abs/1903.10000 (2019).

[196] *TLC Trip Record Data*. `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`.

[197] Ankit Toshniwal et al. "Storm@twitter". In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pp. 147–156.

[198] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. "SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pp. 1153–1170.

[199] *TSSX Library*. `https://github.com/goldsborough/tssx`.

[200] Philipp Unterbrunner, Georgios Giannikis, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. "Predictable Performance for Unpredictable Workloads". In: *PVLDB* 2.1 (2009), pp. 706–717.

[201] David Vengerov, Andre Cavalheiro Menck, Mohamed Zaït, and Sunil Chakkappen. "Join Size Estimation Subject to Filter Conditions". In: *PVLDB* 8.12 (2015), pp. 1530–1541.

[202] * Dimitri Vorona, Andreas Kipf, Thomas Neumann, and Alfons Kemper. "DeepSPACE: Approximate Geospatial Query Processing with Deep Learning". In: *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2019, Chicago, IL, USA, November 5-8, 2019*, pp. 500–503.

[203] Kai Wei. *How We Built Uber Engineering's Highest Query per Second Service Using Go*. 2016. URL: https://eng.uber.com/go-geofence/ (visited on 11/12/2019).

[204] * Christian Winter, Andreas Kipf, Thomas Neumann, and Alfons Kemper. "GeoBlocks: A Query-Driven Storage Layout for Geospatial Data". In: *CoRR* abs/1908.07753 (2019).

[205] Lucas Woltmann, Claudio Hartmann, Maik Thiele, Dirk Habich, and Wolfgang Lehner. "Cardinality estimation with local deep learning models". In: *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019*, 5:1–5:8.

[206] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. "Towards a Learning Optimizer for Shared Clouds". In: *PVLDB* 12.3 (2018), pp. 210–222.

[207] Wentao Wu, Jeffrey F. Naughton, and Harneet Singh. "Sampling-Based Query Re-Optimization". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pp. 1721–1736.

[208] Yi-Leh Wu, Divyakant Agrawal, and Amr El Abbadi. "Applying the Golden Rule of Sampling for Query Estimation". In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pp. 449–460.

[209] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. "Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pp. 1223–1240.

[210] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. "Simba: Efficient In-Memory Spatial Analytics". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pp. 1071–1085.

[211] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Peter Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. "Deep Unsupervised Cardinality Estimation". In: *PVLDB* 13.3 (2019), pp. 279–292.

[212] Simin You, Jianting Zhang, and Le Gruenwald. "Parallel spatial query processing on GPUs using R-trees". In: *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial@SIGSPATIAL 2013, Nov 4th, 2013, Orlando, FL, USA*, pp. 23–31.

[213] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. "GeoSpark: a cluster computing framework for processing large-scale spatial data". In: *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*, 70:1–70:4.

[214] Xiaohui Yu, Calisto Zuzarte, and Kenneth C. Sevcik. "Towards estimating the number of distinct value combinations for a set of attributes". In: *Proceedings of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005*, pp. 656–663.

[215] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Cláudio T. Silva, and Juliana Freire. "GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons". In: *PVLDB* 11.3 (2017), 352–365.

[216] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Spark: Cluster Computing with Working Sets". In: *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*.

[217] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. "Discretized streams: fault-tolerant streaming computation at scale". In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, Farmington, PA, USA, November 3-6, 2013*, pp. 423–438.

[218] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J. Smola. "Deep Sets". In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pp. 3391–3401.

[219] Mohamed Zaït, Sunil Chakkappen, Suratna Budalakoti, Satyanarayana R. Valluri, Ramarajan Krishnamachari, and Alan Wood. "Adaptive Statistics in Oracle 12c". In: *PVLDB* 10.12 (2017), pp. 1813–1824.

[220] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. "The End of a Myth: Distributed Transactions Can Scale". In: *CoRR* abs/1607.00655 (2016).

[221] Tilmann Zäschke, Christoph Zimmerli, and Moira C. Norrie. "The PH-tree: a space-efficient storage structure and multi-dimensional index". In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pp. 397–408.

[222] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. "Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes". In: *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pp. 1567–1581.

[223] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. "SuRF: Practical Range Query Filtering with Fast Succinct Tries". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pp. 323–336.

[224] Jianting Zhang and Simin You. "Speeding up large-scale point-in-polygon test based spatial join on GPUs". In: *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial@SIGSPATIAL 2012, Redondo Beach, CA, USA, November 6, 2012*, pp. 23–32.

[225] Geraldo Zimbrao and Jano Moreira de Souza. "A Raster Approximation For Processing of Spatial Joins". In: *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pp. 558–569.