TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik
Lehrstuhl für Datenbanksysteme

DOCTORAL THESIS

Modern SQL for Knowledge Discovery and Dataset Versioning

Maximilian E. Schüle

TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik

Modern SQL for Knowledge Discovery and Dataset Versioning

Maximilian E. Schüle

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. Helmut Seidl
Prüfer der Dissertation:	1.	Prof. Alfons Kemper, Ph.D.
	2.	Prof. Dr. Thomas Neumann
	3.	Prof. Dr. Torsten Grust

Die Dissertation wurde am 13.12.2021 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.05.2022 angenommen.

Abstract

In order to move computation close to the data inside database systems, this thesis focuses on the SQL integration of data analysis and machine learning algorithms as well as on native support for dataset versioning.

For the integration of data analysis, we implement clustering, graph mining and association rule algorithms in SQL using recursive tables and scripting languages. Recursive tables within HyPer and Umbra also allow to express gradient descent, a major building block when training a model as used in machine learning. We generate code for automatic differentiation in order to automate the derivation of loss functions as required for gradient descent. Automatically deriving a loss function, expressed as SQL lambda function, accelerates computation linearly by the number of cached subexpressions. To further reduce the time needed for training, automatic differentiation within a dedicated gradient descent operator allows to off-load training to GPU.

Based on the presented extensions, we design a meta-language for machine learning to be translated to either Python or SQL. But as most of its operations relied on array data, we extend Umbra by an ArrayQL interface to treat relations as arrays.

For dataset versioning, we propose a layer on top of existing database systems using an additional table for version management, and within code-generating database systems, a data type and a table scan operator that incorporate versioning.

Zusammenfassung

Um Anwendungslogik direkt auf den Daten im Datenbanksystem zu ermöglichen, konzentriert sich diese Arbeit auf die SQL-Integration von Algorithmen für die Datenanalyse und maschinelles Lernen sowie auf die native Unterstützung der Versionierung von Datensätzen.

Für die Integration der Datenanalysealgorithmen implementieren wir Clustering-, Graph-Mining- und Assoziationsregel-Algorithmen in SQL unter Verwendung rekursiver Tabellen und Skriptsprachen. Rekursive Tabellen in HyPer und Umbra erlauben auch Gradientenabstieg in SQL auszudrücken, ein wichtiger Baustein im Bereich des maschinellen Lernens für das Trainieren eines Modells. Um das Ableiten der Verlustfunktion, wie für den Gradientenabstieg benötigt, zu automatisieren, generieren wir Code für automatisches Differenzieren. Das automatisierte Ableiten der Verlustfunktion, ausgedrückt als SQL-Lambda-Funktion, beschleunigt die Laufzeit linear mit der Anzahl wiederverwendeter Teilausdrücke. Um das Trainieren weiter zu beschleunigen, erlaubt ein dedizierter Operator für den Gradientenabstieg mit automatischer Differenzierung, das Training auf die GPU auszulagern.

Basierend auf den vorgestellten Erweiterungen entwerfen wir eine Metasprache für maschinelles Lernen, die entweder zu Python oder zu SQL übersetzt werden kann. Da die meisten Operationen Daten aggregiert zu Feldern erwarten, erweitern wir Umbra um eine ArrayQL-Schnittstelle, um Relationen nativ als Matrizen zu behandeln.

Für die Versionierung von Datensätzen schlagen wir eine Schicht über dem bestehenden Datenbanksystem vor, die eine zusätzliche Tabelle für die Versionsverwaltung verwendet, und innerhalb von codegenerierenden Datenbanksystemen einen Datentyp und einen Tabellenscan-Operator, die Versionierung unterstützen.

Contents

Abstract **iii**

Zusammenfassung **v**

Preface **xi**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Scope . . . 1
1.3 Contributions . . . 2
1.4 Outline . . . 2

2 Related Work **3**
2.1 Database Scripting Languages . . . 3
2.2 Dedicated Machine Learning Tools . . . 4
2.3 Automatic Differentiation . . . 4
2.4 Database Systems and Machine Learning . . . 4
2.5 GPU Acceleration . . . 5
2.6 Array Extensions for Database Systems . . . 5
2.7 Array Database Systems . . . 6
2.8 Database Versioning . . . 6
2.8.1 Temporal Databases . . . 6
2.8.2 Version Control: Git . . . 7
2.8.3 Versioning in DBMS . . . 7

3 Tensor Data Type in SQL **9**
3.1 Slice . . . 10
3.2 Transpose . . . 10
3.3 Addition, Subtraction, Scalar/Hadamard Product . . . 11
3.4 Inner Product . . . 11
3.5 Sparse Tensors . . . 11
3.6 Matrix Usage for Linear Regression . . . 12
3.7 Evaluation . . . 13
3.7.1 Gram Matrix Computation . . . 13
3.7.2 Linear Regression Using Equation Systems . . . 13
3.8 Conclusion . . . 14

4 SQL for Data Analysis: HyPerScript and Recursive Tables **15**
4.1 HyPerScript . . . 16
4.1.1 HyPerScript with Tensor Operations . . . 16
4.1.2 HyPerScript for Business Transactional Applications . . . 17
4.2 SQL for Data Analysis . . . 18
4.2.1 PageRank . . . 18
4.2.2 Clustering . . . 19
4.2.3 Association Rule Analysis . . . 21
4.3 Evaluation . . . 22
4.4 Conclusion . . . 24

5 In-Database Machine Learning **25**
5.1 In-Database Gradient Descent . . . 26
5.1.1 Mini-Batch Gradient Descent . . . 26
5.1.2 Machine Learning Pipeline in SQL . . . 27
5.2 Database Operators for Machine Learning . . . 29
5.2.1 Code Generation . . . 29
5.2.2 Automatic Differentiation . . . 29
5.2.3 Gradient Descent Operator . . . 31
Operator Design . . . 31
Implementation . . . 32
Pipelining and Parallelism . . . 32
5.3 Neural Network . . . 34
5.3.1 Neural Network in SQL-92 . . . 34
5.3.2 Neural Network with an Array Data Type . . . 34
5.3.3 Training a Neural Network . . . 35
5.4 Evaluation . . . 37
5.4.1 Automatic Differentiation in Umbra . . . 37
5.4.2 Gradient Descent Operator in HyPer . . . 40
5.4.3 Pipelining within HyPer . . . 43
5.5 Conclusion . . . 44

6 Multi-GPU Gradient Descent **45**
6.1 Kernel Implementations . . . 45
6.1.1 Linear Regression . . . 46
6.1.2 Neural Network . . . 46
6.1.3 Multiple Learners per GPU . . . 47
6.2 Synchronisation Methods . . . 47
6.2.1 Synchronised Iterations . . . 47
6.2.2 Worker Threads with Global Updates (Bulk Synchronous Parallel) . . . 48
6.2.3 Worker Threads with Local Models (Model Average) . . . 48
6.3 Evaluation . . . 50
6.3.1 Linear Regression . . . 50
Throughput vs. Statistical Efficiency . . . 50
Multiple Learners per GPU . . . 51
Scalability . . . 52
6.3.2 Neural Network . . . 52
Throughput vs. Statistical Efficiency . . . 53
Scalability . . . 55
Time/Tuples-to-loss . . . 55
6.3.3 End-to-End Analysis . . . 55
6.4 Conclusion . . . 55

7 ML2SQL: Compiling a Declarative Meta-Language for ML to SQL **57**
7.1 MLearn and the ML2SQL Compiler . . . 58
7.1.1 Language Specification . . . 58
7.1.2 Target Language . . . 59
7.1.3 Example . . . 60
7.2 Evaluation . . . 61
7.3 Demonstration . . . 61
7.4 Conclusion . . . 62

8 ArrayQL **63**
8.1 In-Database Integration . . . 64
8.1.1 Architecture . . . 65
8.1.2 Array Representation . . . 65
8.1.3 Interaction with SQL . . . 65
8.2 ArrayQL Algebra . . . 66
8.2.1 Rename . . . 66

8.2.2 Function Application . . . 66
8.2.3 Filter . . . 66
8.2.4 Index Manipulation: Shift and Rebox . . . 67
8.2.5 Fill . . . 67
8.2.6 Combining and Joining . . . 67
Combine . . . 67
Inner Join . . . 68
8.2.7 Reduce for Aggregations . . . 68
8.3 ArrayQL Grammar . . . 68
8.3.1 Data Definition Language . . . 68
8.3.2 Data Query Language . . . 69
8.3.3 Data Modification Language . . . 69
8.4 Application of ArrayQL . . . 70
8.4.1 Geo-Temporal Data . . . 70
8.4.2 Linear Algebra with ArrayQL . . . 70
Scalar Operations . . . 71
Transpose and Slice . . . 71
Matrix Multiplication . . . 72
Implemented Table Functions and Short-Cuts . . . 72
Application . . . 72
8.4.3 Logical Optimisations . . . 73
Optimisation Steps . . . 74
Cost-Based Query Plan Reordering . . . 74
8.5 Evaluation . . . 75
8.5.1 Linear Algebra . . . 75
Matrix Algebra . . . 75
Linear Regression . . . 76
8.5.2 Array Operations . . . 77
New York Taxi Data . . . 77
Random Data . . . 79
SS-DB Data . . . 80
8.6 Conclusion . . . 81

9 Versioning in Main-Memory Database Systems 83
9.1 MusaeusDB: Versioning using SQL . . . 85
9.1.1 init . . . 86
9.1.2 checkout . . . 86
9.1.3 commit . . . 86
9.2 Versioning in Code-Generating Database Systems . . . 87
9.3 SQL Extension . . . 88
9.4 Demonstration Setup . . . 89
9.5 TardisBenchmark . . . 89
9.5.1 MediaWiki Schema and Wikipedia Data . . . 90
9.5.2 Operations . . . 90
9.5.3 Storage and Compression . . . 91
9.6 Versioning Data Type in SQL . . . 92
9.6.1 Umbra Integration . . . 92
9.6.2 SQL Integration . . . 93
9.7 Evaluation . . . 93
9.7.1 MusaeusDB . . . 93
9.7.2 TardisBenchmark . . . 94
Insert . . . 94
Space Consumption . . . 96
Retrieve First . . . 96
Retrieve Latest . . . 97
9.7.3 Versioning Data Type . . . 97
9.8 Conclusion . . . 98

10 Conclusion and Future Work **99**
10.1 Conclusion 99
10.2 Future Work 100

A SQL for Data Analysis: Queries **101**
A.1 HyPer 101
A.2 PostgreSQL 104

B In-Database Machine Learning: Scripts **107**

Bibliography **111**

Preface

Parts of the following chapters already appeared in the listed publications.

Chapter 3, 4

[122] Maximilian E. Schüle, Linnea Passing, Alfons Kemper, and Thomas Neumann. "Ja-(zu-)SQL: Evaluation einer SQL-Skriptsprache für Hauptspeicherdatenbanksysteme". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings*. Ed. by Torsten Grust, Felix Naumann, Alexander Böhm, Wolfgang Lehner, Theo Härder, Erhard Rahm, Andreas Heuer, Meike Klettke, and Holger Meyer. Vol. P-289. LNI. Gesellschaft für Informatik, Bonn, 2019, pp. 107–126. DOI: 10.18420/btw2019-08. URL: https://doi.org/10.18420/btw2019-08.

Chapter 3, 5

[124] Maximilian E. Schüle, Frédéric Simonis, Thomas Heyenbrock, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. "In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings*. Ed. by Torsten Grust, Felix Naumann, Alexander Böhm, Wolfgang Lehner, Theo Härder, Erhard Rahm, Andreas Heuer, Meike Klettke, and Holger Meyer. Vol. P-289. LNI. Gesellschaft für Informatik, Bonn, 2019, pp. 247–266. DOI: 10.18420/btw2019-16. URL: https://doi.org/10.18420/btw2019-16.

Chapter 5, 6

[121] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. "In-Database Machine Learning with SQL on GPUs". In: *SSDBM 2021: 33rd International Conference on Scientific and Statistical Database Management, Tampa, FL, USA, July 6-7, 2021*. Ed. by Qiang Zhu, Xingquan Zhu, Yicheng Tu, Zichen Xu, and Anand Kumar. ACM, 2021, pp. 25–36. DOI: 10.1145/3468791.3468840. URL: https://doi.org/10.1145/3468791.3468840.

Chapter 7

[114] Maximilian E. Schüle, Matthias Bungeroth, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. "MLearn: A Declarative Machine Learning Language for Database Systems". In: *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning, DEEM@SIGMOD 2019, Amsterdam, The Netherlands, June 30, 2019*. Ed. by Sebastian Schelter, Neoklis Polyzotis, Stephan Seufert, and Manasi Vartak. ACM, 2019, 7:1–7:4. DOI: 10.1145/3329486.3329494. URL: https://doi.org/10.1145/3329486.3329494.

[115] Maximilian E. Schüle, Matthias Bungeroth, Dimitri Vorona, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. "ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python". In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. Ed. by Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi. OpenProceedings.org, 2019, pp. 562–565. DOI: 10.5441/002/edbt.2019.56. URL: https://doi.org/10.5441/002/edbt.2019.56.

Chapter 8

[116] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. "ArrayQL for Linear Algebra within Umbra". In: *SSDBM 2021: 33rd International Conference on Scientific and Statistical Database Management, Tampa, FL, USA, July 6-7, 2021*. Ed. by Qiang Zhu, Xingquan Zhu, Yicheng Tu, Zichen Xu, and Anand Kumar. ACM, 2021, pp. 193–196. DOI: 10.1145/3468791.3468838. URL: https://doi.org/10.1145/3468791.3468838.

[117] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. "ArrayQL Integration into Code-Generating Database Systems". In: *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*. Ed. by Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang. OpenProceedings.org, 2022, 1:40–1:51. DOI: 10.5441/002/edbt.2022.04. URL: https://doi.org/10.5441/002/edbt.2022.04.

Chapter 9

[67] Lukas Karnowski, Maximilian E. Schüle, Alfons Kemper, and Thomas Neumann. "Umbra as a Time Machine". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2021), 19. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 13.-17. September 2021, Dresden, Germany, Proceedings*. Ed. by Kai-Uwe Sattler, Melanie Herschel, and Wolfgang Lehner. Vol. P-311. LNI. Gesellschaft für Informatik, Bonn, 2021, pp. 123–132. DOI: 10.18420/btw2021-06. URL: https://doi.org/10.18420/btw2021-06.

[119] Maximilian E. Schüle, Lukas Karnowski, Josef Schmeißer, Benedikt Kleiner, Alfons Kemper, and Thomas Neumann. "Versioning in Main-Memory Database Systems: From MusaeusDB to TardisDB". In: *Proceedings of the 31st International Conference on Scientific and Statistical Database Management, SSDBM 2019, Santa Cruz, CA, USA, July 23-25, 2019*. Ed. by Carlos Maltzahn and Tanu Malik. ACM, 2019, pp. 169–180. DOI: 10.1145/3335783.3335792. URL: https://doi.org/10.1145/3335783.3335792.

[123] Maximilian E. Schüle, Josef Schmeißer, Thomas Blum, Alfons Kemper, and Thomas Neumann. "TardisDB: Extending SQL to Support Versioning". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2021, pp. 2775–2778. DOI: 10.1145/3448016.3452767. URL: https://doi.org/10.1145/3448016.3452767.

Further Publications

In addition, the author of this thesis contributed to the following publications.

[61] Nina Hubig, Linnea Passing, Maximilian E. Schüle, Dimitri Vorona, Alfons Kemper, and Thomas Neumann. "HyPerInsight: Data Exploration Deep Inside HyPer". In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*. Ed. by Ee-Peng Lim, Marianne Winslett, Mark Sanderson, Ada Wai-Chee Fu, Jimeng Sun, J. Shane Culpepper, Eric Lo, Joyce C. Ho, Debora Donato, Rakesh Agrawal, Yu Zheng, Carlos Castillo, Aixin Sun, Vincent S. Tseng, and Chenliang Li. ACM, 2017, pp. 2467–2470. DOI: 10.1145/3132847.3133167. URL: https://doi.org/10.1145/3132847.3133167.

[104] Magdalena Pröbstl, Philipp Fent, Maximilian E. Schüle, Moritz Sichert, Thomas Neumann, and Alfons Kemper. "One Buffer Manager to Rule Them All: Using Distributed Memory with Cache Coherence over RDMA". In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021, Copenhagen, Denmark, August 16, 2021*. Ed. by Rajesh Bordawekar and Tirthankar Lahiri. 2021, pp. 17–26. URL: http://www.adms-conf.org/2021-camera-ready/probstl_adms21.pdf.

[109] Josef Schmeißer, Maximilian E. Schüle, Viktor Leis, Thomas Neumann, and Alfons Kemper. "B^2-Tree: Cache-Friendly String Indexing within B-Trees". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2021), 19. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 13.-17. September 2021, Dresden, Germany, Proceedings*. Ed. by Kai-Uwe Sattler, Melanie Herschel, and Wolfgang Lehner. Vol. P-311. LNI. Gesellschaft für Informatik, Bonn, 2021, pp. 39–58. DOI: 10.18420/btw2021-02. URL: https://doi.org/10.18420/btw2021-02.

[113] Maximilian Schüle, Pascal Schliski, Thomas Hutzelmann, Tobias Rosenberger, Viktor Leis, Dimitri Vorona, Alfons Kemper, and Thomas Neumann. "Monopedia: Staying Single is Good Enough - The HyPer Way for Web Scale Applications". In: *Proc. VLDB Endow.* 10.12 (2017), pp. 1921–1924. DOI: 10.14778/3137765.3137809. URL: http://www.vldb.org/pvldb/vol10/p1921-schuele.pdf.

[118] Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. "Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL". In: *SSDBM 2020: 32nd International Conference on Scientific and Statistical Database Management, Vienna, Austria, July 7-9, 2020*. Ed. by Elaheh Pourabbas, Dimitris Sacharidis, Kurt Stockinger, and Thanasis Vergoulis. ACM, 2020, 6:1–6:12. DOI: 10.1145/3400903.3400915. URL: https://doi.org/10.1145/3400903.3400915.

[120] Maximilian E. Schüle, Alex Kulikov, Alfons Kemper, and Thomas Neumann. "ARTful Skyline Computation for In-Memory Database Systems". In: *New Trends in Databases and Information Systems - ADBIS 2020 Short Papers, Lyon, France, August 25-27, 2020, Proceedings*. Ed. by Jérôme Darmont, Boris Novikov, and Robert Wrembel. Vol. 1259. Communications in Computer and Information Science. Springer, 2020, pp. 3–12. DOI: 10.1007/978-3-030-54623-6_1. URL: https://doi.org/10.1007/978-3-030-54623-6_1.

[125] Maximilian E. Schüle, Dimitri Vorona, Linnea Passing, Harald Lang, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. "The Power of SQL Lambda Functions". In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. Ed. by Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi. OpenProceedings.org, 2019, pp. 534–537. DOI: 10.5441/002/edbt.2019.49. URL: https://doi.org/10.5441/002/edbt.2019.49.

Chapter 1

Introduction

In a multitier architecture, the traditional wisdom is to use a database system as data storage only. Within its responsibility, the database system guarantees the ACID properties out of atomicity, consistency, isolation and durability for a transaction. For querying the data, database systems offer the declarative language SQL, allowing the user to specify *what* to do, rather than caring about the implementation nor its optimisation. Furthermore, SQL, as a standardised language, ensures the exchangeability of database systems in theory.

In this thesis, we argue that modern hardware allows database systems to take over more computation. To move computation to the data inside a database system, we destine SQL for specifying user-defined algorithms, since SQL is already present and inherits the benefits of a declarative language. In the remainder of this chapter, we present the motivation for in-database computations, that is to reduce the transfer of data, define this thesis' scope and contribution to in-database machine learning and dataset versioning and close by an outline.

1.1 Motivation

Data warehouses form a common example, where relational database systems are used for online transactional processing (OLTP) and online analytical processing (OLAP). OLTP means an update-heavy workload, as occurs when customers are ordering items, whereas OLAP mostly depends on read-only queries that analyse and aggregate large datasets such as the number of sold items per day. In the past, transactional and analytical queries were processed separately on for each use-case specialised system, requiring an extraction, transform, load (ETL) process in-between the data and application layer. With modern hardware, the size of main-memory increased and systems for hybrid transactional/analytical processing (HTAP) such as HyPer [69] could be developed, which allow combining both use cases. Nevertheless, the ETL process still is an issue. With the arising complexity of data analysis and machine learning algorithms, the data gets extracted and then analysed externally [144].

1.2 Scope

The overall goal of this thesis is to broaden the usage of database systems for computational tasks. This includes additional operators, data types and operations addressed within SQL to facilitate moving computation to the data stored inside database systems, as well as managing different versions of a database.

As many tasks require an array data type as part of SQL, this thesis introduces such a data type for the code-generating database systems HyPer and Umbra together with operations for linear algebra. This thesis then continues with data analysis algorithms described in the SQL scripting language extensions, namely *HyPerScript* and *PL/pgSQL*, and compares their performance to integrated database operators and recursive SQL tables.

We continue with recursive SQL used for gradient descent, on which modern machine learning algorithms are based. This allows formulating a complete machine learning pipeline within SQL including data preprocessing, continuous training and sampling batches for gradient descent with manually derived loss functions. To eliminate the need for manually deriving functions, we integrate automatic differentiation of SQL lambda functions into a code-generating database system, i.e., Umbra. Based on automatic differentiation, we propose a dedicated operator for gradient descent to decrease the time needed to train a model,

which we integrate into HyPer and Umbra. Such a materialising operator allows off-loading to GPU, which accelerates vectorised computations as needed for gradient descent. We suggest the implementation of fine-grained learners to exploit the maximal throughput on a GPU.

On top of database systems, we propose a declarative meta-language for machine learning, whose compiler translates statements for either a Python or an SQL backend. For the SQL backend, we make use of the proposed database extensions out of tensors, gradient descent and procedural statements within a scripting language. However, most of the operations are based on the tensor data type, which requires aggregation of tables into arrays beforehand. To eliminate the need for such aggregations, we come up with ArrayQL, a language extension for relational database systems that interprets tables as arrays and translates array statements into operator plans.

Another aspect concerns version management of datasets. Relational database systems also dominate within online encyclopaedias such as Wikipedia. We argue that native support within database systems for versioning will eliminate the need for an additional layer. Besides, persisting a certain database state without extracting the data as a snapshot supports the reproducibility of experiments, since the same data leads to the same results.

1.3 Contributions

In particular, this thesis contributes to:

- *data analysis* by algorithms described in recursive SQL or a scripting language, and moreover, an evaluation of their performance within modern database systems, namely HyPer and Umbra, and a traditional one, i.e., PostgreSQL with PL/pgSQL and MADlib;
- *in-database machine learning* by gradient descent using a recursive table and machine learning pipelines including preprocessing and continuous training expressed in SQL, automatic differentiation as an operator within a code-generating database system to facilitate gradient descent, the architectural blueprint for gradient descent as an operator within SQL, customised GPU kernel implementations for model training, an SQL array data type with operations for tensor algebra, and an intermediate machine learning language that translates into SQL or Python scripts using common libraries;
- *array database systems* by a relational representation of arrays within a relational, code-generating database system, i.e., Umbra, to substitute array database systems, an extension to support linear algebra and an evaluation that compares the relational representation to common array database systems;
- *dataset versioning* by supporting versions over multiple tables using a tool on top of relational database systems, an integration into code-generating database systems, an SQL versioning data type supporting delta-compression, an SQL extension for database versioning that considers referential integrity and an evaluation to compare different storage approaches.

1.4 Outline

This thesis is structured as follows: Chapter 2 summarises related work on machine learning, matrix algebra and dataset versioning in the context of relational database systems. Chapter 3 extends an SQL array data type for tensor algebra. We use this data type in conjunction with customised operators and recursive SQL tables for data analysis in Chapter 4. Based on recursive SQL statements, Chapter 5 proposes a complete machine learning pipeline within database systems including operators for automatic differentiation. A materialising operator for gradient descent allows off-loading to GPU as discussed in Chapter 6. Based on these extensions, Chapter 7 describes the *ML2SQL* compiler to translate the meta-language *MLearn* into SQL. In order to interpret tables as arrays directly, Chapter 8 describes the integration of ArrayQL into Umbra. Chapter 9 presents dataset versioning on top of relational and inside of code-generating database systems. Chapter 10 concludes this thesis by giving an outlook on required SQL statements and possible extensions for traditional database systems.

Chapter 2

Related Work

This chapter is a consolidation of related work sections in [117, 119, 122, 123, 124].

This chapter describes scripting languages similar to *HyPerScript*, state-of-the-art machine learning tools, the current status of in-database machine learning and data analytics, array handling within database systems and dataset versioning.

2.1 Database Scripting Languages

The open-source application *GNU R*[1], which is widely used for statistical analysis, offers modules, which can be controlled via a separate command line. These modules are comparable to the operators offered in *HyPer* [102] because both (the modules and the operators) represent building blocks with hard-coded functionality. If one wants to add new modules or change the existing ones, new packages (*GNU R*) or new operators (*HyPer*) have to be developed in C++. The alternative to a new development are functions definable in R (see Listing 2.1). for which procedural language constructs such as loops and other control structures are available. The counterpart for defining functions in database systems are *stored procedures* with *HyPerScript* or *PL/pgSQL* as presented in Chapter 4. Besides *GNU R*, other tools, such as *Julia*[2], exist, which, like our *HyPerScript*, generates LLVM (Low Level Virtual Machine) code for high-performance data analysis.

```
1 insertuntil <- function(count) {
2    r <- c();
3    for(i in 1:count) { rand=sample(1:100,1); if(rand!=13) r<-c(r,rand); };
4    return (r);
5 };
```

LISTING 2.1: Example of a function in *GNU R* that returns a vector of up to `count` random numbers between `0` and `100` without the number `13`.

Crotty et al. [29] propose a comparable approach to functions compiled with *HyPerScript*. Their approach compiles, combines and optimises arbitrary, *user-defined functions* (UDFs) into LLVM code to use vector instructions and pipelining on datasets. A common feature is the compilation of user-defined functions to LLVM code (although the suffix "Script" in *HyPerScript* incorrectly leads to the assumption that functions get interpreted). To decrease the runtime of queries in PostgreSQL, the approach in [23, 24] compiles SQL queries to LLVM code. Another modification of PostgreSQL is the extension of PL/pgSQL by higher order functions [49, 50]. The integration of algebraic data types into relational database systems [46] poses another challenge as described in Chapter 3.

SAP HANA with *SQLScript* [12] offers a programming language comparable to *HyPerScript*. *SQLScript* extends SQL by procedural and functional statements, supports *MapReduce* on analytical queries and extends recursion for graph analysis. As an alternative to procedural extensions, *FunSQL* [13] extends SQL with functional constructs. The extension, similar to this work, intends to shift application logic into database servers.

To move computation into database servers, a meta-language as presented in Chapter 7, that translates to SQL, is helpful. Another example is *Ferry* [48, 111], a meta-language with the constructs *for*, *where*, *group by*, *order by* and *return*. *Ferry* is first generated from common scripting or programming languages and then translated into SQL.

[1] R `http://www.r-project.org/`
[2] Julia `http://julialang.org/`

2.2 Dedicated Machine Learning Tools

Dedicated machine learning tools, such as Google's TensorFlow [1] and Theano [5], support automatic differentiation to compute the gradient of loss functions as part of their core functionality. They transform data in tensors until it is possible to apply a loss function. As they are external to the database system, they require the extraction and transfer of data. To integrate gradient descent, we only need the automatic differentiation component of the tools. TensorFlow is based on tensors connected to graphs and their evaluation. Theano [5] is a linear algebra compiler for Python. Its graph representation of mathematical expressions allows symbolic differentiation for prototyping models to be optimised by gradient descent. Designed specifically for neural networks on top of Theano and TensorFlow as a user-friendly interface, Keras[3] broadens the application area of machine learning tools.

To cover the life-cycle of machine learning pipelines, automatic machine learning (AutoML) tools such as *Alpine Meadow* [127], Lara [78] or *ABC* [59] assist in data preprocessing as well as finding the best hyper-parameters within different configuration setups. Likewise, *POS* [139] allows online learning on workflows with scientific data. When we integrate a machine learning pipeline in SQL, we adapt the idea of continuous deployment of machine learning pipelines [36]. The idea is based on an architecture that monitors the input stream and avoids complete retraining by sampling batches.

2.3 Automatic Differentiation

Differentiation methods can be divided into symbolic, numeric (approximating the derivative) or automatic differentiation; that again can be split into forward and reverse mode [7] depending on the direction in which the chain rule is applied. Examples for deep learning frameworks for neural networks who support automatic differentiation are Mxnet [26] and Pytorch[4]. They also provide tensor computation and gradient descent as part of their core features. As existing gradient differentiation tools do not allow for tensor derivatives, Laue et al. [79] proposed a framework for automatically differentiating higher-order matrices and tensors. This is based on the Ricci calculus and allows both forward and backward mode. Within Umbra, we generate code for reverse mode automatic differentiation.

2.4 Database Systems and Machine Learning

The MADlib [54] analytics library extends existing database systems such as PostgreSQL to perform data mining and statistical analysis. It provides a matrix data type and table functions for linear and logistic regression, which we use as a baseline. EmptyHeaded [2] and HyPer [102] integrate data mining algorithms into database systems as operators. Whereas EmptyHeaded compiles algorithms and relational algebra together, HyPer provides data mining operators as part of its relational algebra. This work extends its relational algebra for machine learning. Another architecture for scalable analytics is XDB [14] that unifies transaction guarantees from database systems, partitioning, and adaptive parallelisation strategies for analytical workloads. Tupleware [30, 31] is built for distributed analytics and generates code for distributed execution. Vizdom [32] is an interface for Tupleware for visually sketching machine learning workflows. Both approaches also intend to combine the benefits of database and analytics systems.

Bridging the gap solutions try to combine the advantages of SQL-like declarative languages but defining their own syntax. BUDS [44] is a declarative language for statistical analysis, which translates to SQL. The work claims the importance of array data types in database systems, which perform better than table-based array structures. In Chapter 3, we extend this work by the evaluation of array/tensor data types in main-memory database systems. Another declarative language comparable to the one shown in Chapter 7 is MLog [85], which can be compiled to an executable program using the TensorFlow API to provide a translation to SQL.

[3] https://keras.io/
[4] http://pytorch.org/

For the vision of a declarative framework for machine learning, Kaoudi et al. [63] adapt database systems' optimisers for gradient descent computation. Using a cost function it chooses a computation plan out of stochastic, batch, or mini-batch gradient descent. Another notable automatic differentiation framework is Hogwild [105]. It presents an approximative way of lock-free parallel stochastic gradient descent without synchronisation.

Independent machine learning systems that resemble database systems are Bismarck [42], Rafiki [136] or SystemML [18] with its own declarative programming language and its successor SystemDS [17]. Integration of machine learning pipelines inside of database systems would allow end-to-end machine learning and would inherit benefits such as query optimisation and recovery by design [77, 95]. As a layer above of database systems, LMFAO [108] learns models on pre-aggregated data using SQL. Raven [64] supports machine learning tasks in Microsoft SQL Server using a unified intermediate representation.

The work of Jankov et al. [62] states, that complete integration is possible by extending SQL with additional recursive statements. As recursive tables perform better than procedural SQL extensions, Duta et al. [39, 40] and Hirn et al. [56, 58] transform PL/pgSQL statements into recursive *common table expressions* (CTEs). In this thesis, we avoid procedural constructs for machine learning by using recursive statements for data mining algorithms (Chapter 4) and for gradient descent (Chapter 5) inside code-generating database systems. Similarly to this approach, Blacher et al. [16] perform logistic regression in SQL using a recursive table and matrix algebra to derive the gradient symbolically. As a novelty, they map procedural constructs from Python to CTEs in SQL.

2.5 GPU Acceleration

Research on GPU support for database systems focused on hybrid operator plans [27] and its performance with hybrid transactional and analytical workloads [64]. *Crossbow* [75] is a machine learning framework, written in Java, that maintains and synchronises local models for independent learners that call C++ functions to access NVIDIA's deep neural network library *cuDNN*[5]. We rely on the study when adjusting batch sizes for GPUs and synchronising multiple workers (Chapter 6).

The LLVM compiler framework, often used for code-generation within database engines [57], also offers just-in-time compilation for NVIDIA's Compute Unified Device Architecture (CUDA) [92]. So far, code-compilation for GPU accelerates genetic computations [35, 84], but also allows compiling for heterogeneous CPU-GPU clusters [86] as LLVM addresses multiple target architectures. As modern database systems generate code, JIT-compiling to LLVM allows seamless integration of GPU co-processing.

2.6 Array Extensions for Database Systems

MADlib [54] operates on tables in a relational representation, which they call a sparse matrix, as well as on the array data type. Also implemented as a data type inside database systems, Kernert et al. [72] enable native support for linear algebra on dense and sparse matrices. However, enabling a separate data type has the downside of expensive transformations out of tables.

Luo et al. [91] argue that database systems form an excellent platform for linear algebra as this kind of computations can be expressed as a combination of operators within relational algebra. Due to the complexity of writing linear algebra computations in SQL and the overhead of the Volcano-style iterator model [47], they propose adding a vector and matrix data type as database attributes and a small set of SQL language extensions for corresponding operations. Umbra eliminates the overhead of one function call per operator introduced by the Volcano-style iterator model as it generates LLVM code according to the producer-consumer model [99]. This allows pipelined processing, reduces the cost per tuple significantly and achieves nearly the performance of a hard-coded implementation. This thesis benchmarks the performance increase for linear algebra within such a code-generating database system in comparison to traditional (array) database systems. Although Umbra provides a data

[5] https://developer.nvidia.com/cudnn

type to store matrices as a part of relational tables, the motivation in Chapter 8 is to provide an array view on tables.

To allow linear algebra directly on database tables, relational matrix algebra (RMA) [37] extends MonetDB by operators for linear algebra. The linear operations can be addressed in SQL as table functions. But in contrast to Chapter 8, RMA interprets tables as matrices (tabular representation), limited to two-dimensional matrices, and requires a row-ordering as contextual information among linear operations. In contrast, SPORES [137] uses a relational representation for matrices only as an intermediate format to derive optimisations from relational algebra to SystemML. When we base our matrices directly on tables and translate all operations into relational algebra, we earn these optimisations [77] for free.

2.7 Array Database Systems

Introduced in 1997, RasDaMan [6] was the first array database system developed for geospatial data. Even though it supports relational database systems as underlying storage engines, it stores data as binary large objects (BLOBs) similar to a file system. Therefore, querying happens within RasDaMan only, for which it offers the array query language RasQL, including SQL-92 and embedded statements for multi-dimensional data. These statements access arrays as regular expressions and have been incorporated in the SQL/MDA:2019 standard[6] for multi-dimensional fields.

SciDB [34] is the first database system that uses arrays as a first-class data model. Its declarative query language AQL is based on SQL and the array programming language APL. Another array database system is SciQL [147], which uses MonetDB's query engine and storage layout. Binary association tables (BAT), normally used to store columns, hold the array data. This allows one unified query interface to address either SQL tables or arrays. Chapter 8 presents ArrayQL as an array query language with access to SQL tables.

2.8 Database Versioning

Whereas version control for software projects has a long tradition, studies on database systems have mainly focused on temporal databases. This section describes temporal databases and research on full versioning of databases (mostly limited to single datasets) and *Git* as the most popular version control system, which is later used as competitor.

2.8.1 Temporal Databases

Temporal databases bind the validity of tuples to time intervals by offering an additional data type. Until 2011, several extensions of SQL such as *TQuel* [129] and *TSQL2* [130] examined adding time travelling to fetch the database state from a certain date in the past. They tried to be downward compatible with SQL-92 but were not considered for a new SQL standard. Instead, the SQL:2011 standard [76] marks the tuples of a temporal database with two date columns indicating the period of the tuple's validity. Nowadays, most of the commercial database systems support time travelling. For example, *Microsoft SQL Server (MSSQL)*[7], *IBM DB2 10*[8] and *MariaDB*[9] take a time range from two date columns. *PostgreSQL (PSQL)*[10] offers the `tstzrange` type to indicate time instances and an additional history table for past records. Based on the presence of data types for temporal databases, different optimising techniques aim at integrating time travelling inside transaction handling [89] or at indexing tuples for accessing time evolving data efficiently [107]. Beyond relational database systems, studies focus on compressing time evolving data as XML archives [21] or on integrating time travelling inside RDF databases to be addressed with SPARQL [132].

[6] https://www.iso.org/standard/69777.html
[7] www.mssqltips.com/sqlservertip/3680/introduction-to-sql-server-2016-temporal-tables/
[8] www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/
[9] https://mariadb.com/kb/en/library/system-versioned-tables/
[10] https://wiki.postgresql.org/wiki/SQL2011Temporal

2.8.2 Version Control: Git

Git[11] was initially developed for managing large software projects, but can be used for other purposes as collaborative text editing. Internally, commits represent changes; one commit may have multiple preceding ones, forming an acyclic graph. It works decentrally and allows branching and merging for collaborative code development.

Actually, Git is a key-value store for storing any kind of data returning a unique hash identifier, which allows the data to be retrieved. Internally, Git stores three types of objects, *blob*, *tree* and *commit*. A *blob* object represents the content (*binary large object*) that Git has to track, so every object corresponds to one file. Consequently, for every minor edit, a new *blob* object is created. The *tree* objects correspond to the directories and store the identifiers of the contained *blob* or sub-*tree* objects. Finally, the *commit* objects contain a parent one and the *tree* objects.

This so-called *Loose Object Format* would be inefficient if no packing took place. But Git uses delta encoding, a form of taking differences between objects. The optimal differences, adjustable by one variable that sets the maximum length of a delta-chain and one for the number of files to compare, are stored in *packfiles*. This information is useful for adapting delta encoding and for understanding the benchmark results in Section 9.7.

TABLE 2.1: Overview of data versioning tools.

System	Versioning	Querying	Data Model	Branching	Constraints
DataHub [9]	VQL	Thrift API	Relational	Numbers	No
Decibel [93]	VQuel	SQL	Relational	Numbers	No
ForkBase [88]	REST API	None	Key-Value	Names	No
LiteTree	Using Pragma	SQL	Relational	Numbers	No
MusaeusDB [119]	Bash Script	SQL	Relational	Numbers	Yes
OrpheusDB [60]	Python API	SQL	Relational	Numbers	No
RStore [11]	Java API	CQL	Key-Value	Numbers	No
dolt	SQL	SQL	Relational	Names	Yes
TardisDB	SQL	SQL	Relational	Names	Yes

2.8.3 Versioning in DBMS

In contrast to temporal databases, which protocol the validity of each tuple, versioning should log at a higher granularity to preserve the whole database's state. Table 2.1 lists tools for dataset versioning that are similar to database systems.

One example with its versioning language VQL is the *DataHub* [9] platform whose flexible API allows applications such as an SQL interface to be built on top. The core part was highlighted as *Decibel* [93] with the language *VQuel* [25] to manage versions. These systems have in common that they build the application layer above the data storage rather than offering an extended relational algebra that supports versioning. It evaluates different bitmap based storage techniques for creating and merging branches by using a self-created versioning benchmark. The versioning scope is limited to single relations. We take the idea of bitmaps indicating included tuples for each branch of a main-memory database system.

A version control on top of database systems is *OrpheusDB* [143, 60], called a "bolt-on" technique as it works on existing datasets. First, the data to be versioned is loaded from CSV files or from a database system into an extended database schema. Then, every tuple is extended by a record identifier *rid*. A version consists of multiple rids, which are managed in a separate table. The database system itself remains unmodified as OrpheusDB can run on top of any arbitrary database system as long as it provides SQL-92 commands and an array data type. Our SQL prototype in Section 9.1, *MusaeusDB*[12] [74, 119], will extend this work to manage multiple tables.

R-Store [11] is another key-value store based on Apache Cassandra, which also maintains bitmaps to indicate branches. It investigates the trade-off between storage costs, query performance and online updates. It uses delta compression to compress textual documents.

[11] https://git-scm.com/

[12] Musaeus is a contemporary of Orpheus; in honour of *OrpheusDB*.

For unstructured data, *Forkbase* [88] is a tamper-proof data storage system that allows named branches similar to Git and compresses data using similarity graphs.

A key challenge for versioning datasets—postulated in 2015 [10]—is the trade-off between reducing the amount of storage with delta compression while restoring datasets fast enough. To tackle the trade-off, array database systems, designed to host array-like datasets, incorporate versioning techniques as forward or backward delta compression [131].

SciDB [126] decides on a minimum weight spanning tree either to materialise versions for fast recreation or to store the differences for dense as well as sparse array-oriented data. We will use delta compression for text-like data.

Presented in 2018, *LiteTree*[13] is a modification of the file-oriented SQLite database system. It allows access to and modification of any former database state using a version number within pragma statements. It implicitly handles every SQL insert, delete or update statement as another commit, which can be checked out when needed. We also integrate versioning into a relational database system, but with a code-generating engine as the target, we adapt the table scan operator and multi-version concurrency control. We use *LiteTree* as the competitor during our *TardisBenchmark*.

Based on MySQL, *dolt*[14] is a database system with built-in versioning commands in SQL, also introduced in 2018. To identify a branch, they add a keyword `as of` behind each specified table name followed by a commit description, which is the branch name or other type of reference. Although *dolt* does not apply compression techniques, its SQL syntax resembles the one introduced for *TardisDB* in Section 9.3 and it also supports reference key constraints.

[13] https://github.com/aergoio/litetree
[14] https://github.com/dolthub/dolt

Chapter 3

Tensor Data Type in SQL

Parts of this chapter have been published in [124] or are based on a translation of [122].

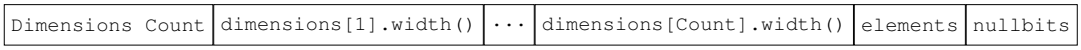


FIGURE 3.1: Internal array representation.

We consider tensor operations necessary for the suitability of database systems for machine learning, for example, to solve linear regression numerically. Therefore *HyPer* and *Umbra* extend their array data type by linear algebra, which allows arrays of integers or floating-point numbers simulating a vector, a matrix or a tensor as an attribute within a database schema. The array data type within HyPer/Umbra is based on the data type for SQL strings whose size is evaluated at runtime. The number of dimensions, each dimension's size, followed by the array elements and a bit vector—indicating each entry's `null` state—are stored in sequential order (see Figure 3.1). An array expression may be part of any arithmetic expression inside of an SQL statement like a regular database attribute. The array operations are applied to attributes as part of the projection in the `SELECT` clause of an SQL query. This includes the following applications:

1. **Algebra**: This includes the tensor product, addition, subtraction and the scalar product, inverting, transposing and exponentiating tensors.

2. **Creation**: Besides creating an array with `ARRAY[<VALUES>]` or `{<VALUES>}`, we need the identity matrix `array_id(<dimension>)` and an array filled with a given value: `array_fill(<VALUES>,<dimensions>)`.

3. **Access**: `array_access(<ARRAY>,<Indices>)` or `ARRAY[<Index>]` accesses an array element. The function to slice `array_slice(<ARRAY>,<Indices>)` allows extracting a subset of an array to split data into a test and training set, for example.

4. **Modification**: After creating an array, modification of an individual element is possible using `array_set(<ARRAY>,<VALUE>,<Indices>)`. The counterpart to slicing mentioned above is the concatenation of arrays `<ARRAY>||<ARRAY>`.

5. **Set Operations**: To identify a subset of an array, a database system should provide set operations like $A \subseteq B$ as `<ARRAY> <@ <ARRAY>`. To find elements in a subset, $a \in A$ can be expressed as `a ANY= A`.

6. **Normalising**: `unnest(<ARRAY>)` as counterpart to array creation decomposes elements into individual tuples.

This chapter describes the implementations of *slice*, *transpose*, *(scalar/Hadamard) product* and *addition/subtraction* on the array data type, and discusses and benchmarks their application for linear regression and on sparse data using gram matrix computation.

3.1 Slice

Slice as an array operation (see Listing 3.1) returns a new array of the same dimensions containing elements of the specified range. We need *slice* to split data into training and test set or for benchmarking purposes. The function `slice` of the corresponding implementation (see Algorithm 1) expects a pointer to the old and the new memory location. The implementation uses a vector of size m (`index`) to calculate the offset within the old location, assigns its content to the new location and increments its pointer iteratively:

$$\begin{aligned} T &\in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_m}, \\ T' &= T_{I_{1a}:I_{1b},\ldots,I_{ma}:I_{mb}} \in \mathbb{R}^{I_{1b}-I_{1a} \times \ldots \times I_{mb}-I_{ma}}, \\ (T')_{i_1 i_2 i_3 \ldots i_m} &= t_{(i_1-I_{1a})(i_2-I_{2a})\ldots(i_m-I_{ma})}. \end{aligned}$$

```
1 create table tensors (a integer[2][3][4], b integer[4][2][1]);
2 insert into tensors values ('{{{1,2,3,4},{5,6,7,8},{9,10,11,12}},
      {{1,2,3,4},{5,6,7,8},{9,10,11,12}}}', '{{{1},{2}},{{3},{4}},{{5},{6}},{{7},{8}}}');
3 select a[2:2][2:3][2:3] from tensors; --[2:2][2:3][2:3]={{{6,7},{10,11}}}
```

LISTING 3.1: Slice.

Algorithm 1 Slice

```
1: function GETOFFSET(position[], size[], m)
2:     offset ← 0
3:     widthproduct ← 1
4:     for i ← m − 1; i ≥ 0; i ← i − 1 do
5:         offset ← offset + position[i] · widthproduct
6:         widthproduct ← widthproduct · size[i]
7:     return offset
8: function INCREMENT(position[], from[], to[], m)
9:     for i ← m − 1; true; i ← i − 1 do
10:        position[i] ← position[i] + 1
11:        if position[i] = to[i] then
12:            position[i] ← from[i]
13:            if i = 0 then return false
14:        elsereturn true
15: function SLICE(∗old, ∗new, m, size[], from[], to[])
16:    index ← from;
17:    while INCREMENT(index,from,to,m) do
18:        ∗new ← ∗(old + GETOFFSET(index, size, m))
19:        new ← new + 1;
```

3.2 Transpose

We define *transpose* as swapping the order of the first two dimensions, with further dimensions being treated as one huge block element. Listing 3.2 produces the transposition T^T of the tensor T as follows:

$$\begin{aligned} T &\in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_m}, \\ T^T &\in \mathbb{R}^{I_2 \times I_1 \times \ldots \times I_m}, \\ (T^T)_{i_1 i_2 i_3 \ldots i_m} &= t_{i_2 i_1 i_3 \ldots i_m}. \end{aligned}$$

```
1 select tensor_transpose(a) from tensors;
2 --{{{1,2,3,4},{1,2,3,4}},{{5,6,7,8},{5,6,7,8}},{{9,10,11,12},{9,10,11,12}}}
```

LISTING 3.2: Tensor transpose.

3.3 Addition, Subtraction, Scalar/Hadamard Product

We handle addition or subtraction elementwise (see Listing 3.3), so the tensors S and T must have identical dimensions. As by definition, this is also the case for the Hadamard product $S \circ T$ (elementwise matrix multiplication), which we expose as SQL operation $\star\star$. Also the scalar product acts elementwise for a given scalar $r \in \mathbb{R}$:

$$\begin{aligned} T, S, T+S, T-S, r \cdot T, T \circ S &\in \mathbb{R}^{I_1 \times \ldots \times I_m}, \\ (T+S)_{i_1 \ldots i_m} &= t_{i_1 \ldots i_m} + s_{i_1 \ldots i_m}, \\ (T-S)_{i_1 \ldots i_m} &= t_{i_1 \ldots i_m} - s_{i_1 \ldots i_m}, \\ (T \circ S)_{i_1 \ldots i_m} &= t_{i_1 \ldots i_m} \cdot s_{i_1 \ldots i_m}, \\ (r \cdot T)_{i_1 \ldots i_m} &= r \cdot t_{i_1 \ldots i_m}. \end{aligned}$$

```
select 2*a+a from tensors;
--{{{3,6,9,12},{15,18,21,24},{27,30,33,36}},{{3,6,9,12},{15,18,21,24},{27,30,33,36}}}
select a**a from tensors;
--{{{1,4,9,16},{25,36,49,64},{81,100,121,144}},{{1,4,9,16},{25,36,49,64},{81,100,121,144}}}
```

LISTING 3.3: Tensor addition and scalar/Hadamard product.

3.4 Inner Product

Matrices like $A \in \mathbb{R}^{m \times o}, B \in \mathbb{R}^{o \times n}$ with equal inner dimensions can be multiplied to create the product $C \in \mathbb{R}^{m \times n}$ by summing up the product of m row elements with n column elements for each entry $c_{ij} = \sum_{k=1}^{o} a_{ik} b_{kj}$. We can generalise this multiplication process to create the inner product of tensors with equal inner dimensions where the new entries are sums of corresponding products (see Listing 3.4):

$$\begin{aligned} T \in \mathbb{R}^{I_1 \times \ldots \times I_m = o}, U &\in \mathbb{R}^{J_1 = o \times \ldots \times J_n}, \\ S = TU &\in \mathbb{R}^{I_1 \times \ldots \times I_{m-1} \times J_2 \times \ldots \times J_n}, \\ s_{i_1 i_2 \ldots i_{m-1} j_2 \ldots j_m} &= \sum_{k=1}^{o} t_{i_1 i_2 \ldots i_{m-1} k} \cdot u_{k j_2 \ldots j_m}. \end{aligned}$$

```
select a*b from tensors;
--{{{{50},{60},{114}},{{140},{178},{220}},{{50},{60},{114}},{{140},{178},{220}}}}
```

LISTING 3.4: Tensor product.

3.5 Sparse Tensors

Beside the representation of tensors as a dense data type, a relational representation $A\{[i, j, a]\}$ identifies each entry $(a_{ij}) \in \mathbb{R}^{I \times J}$ by its indices to avoid storing null values (see Listing 3.5).

```
create table matrixa (i integer, k integer, value float);
create table matrixb (k integer, j integer, value float);
select i,j, sum(a.value*b.value) from matrixa a, matrixb b where a.k=b.k group by i,j;
```

LISTING 3.5: Gram matrix computation using a relational matrix representation.

Between the two possiblities (tensor data type and relational representation) intermediate formats exist like representing each column or row as a tuple, indexed by its column or row identifier with a one-dimensional vector as value (see Listing 3.6).

```
create table matrixa (i integer, value float[]);
create table matrixb (i integer, value float[]);
select i,j, sum(a.value*b.value)
from (select i,row_number() over (partition by b.val) j, unnest(b.val) from matrixa) a,
     (select i,row_number() over (partition by b.val) j, unnest(b.val) from matrixb) b
where a.k=b.k group by i,j;
```

LISTING 3.6: Gram matrix computation using an intermediate representation.

For gram matrix computation, we can also combine column and row relational representation, as we transpose one matrix, for faster results (see Listing 3.7).

```
create table matrix (i integer, value float[]);
select a.i,a.i, a.val*array_transpose(b.val) from matrix a, matrix b;
```

LISTING 3.7: Gram matrix computation using a combined intermediate representation.

3.6 Matrix Usage for Linear Regression

Linear regression approximates (missing) labels for existing attributes using a linear function. Given pairs of attribute and label $(x, y) \in X \subseteq \mathbb{R}^2$, the optimal weights $a, b \in \mathbb{R}$ for simple linear regression $ax + b \approx y$ can be computed using SQL. For multiple attributes $\vec{x} \in \mathbb{R}^m$, we rely on matrix operations to compute the optimal weights $\vec{a} \in \mathbb{R}^m, b \in \mathbb{R}$ for multiple linear regression $\vec{x}^T \cdot \vec{a} + b \approx y$. If we set the gradient equal to zero, we will solve simple and multiple linear regression problems in terms of equation systems such as:

$$0 \stackrel{!}{=} \frac{1}{|X|} \sum_{(x,y)\in X} \nabla(ax + b - y)^2 = \frac{1}{|X|} \sum_{(x,y)\in X} \begin{pmatrix} 2(ax + b - y) \cdot x \\ 2(ax + b - y) \cdot 1 \end{pmatrix}.$$

Using helper variables for the means of attributes $\hat{x} = \sum_{(x,y)\in X} \frac{x}{|X|}$ and labels $\hat{y} = \sum_{(x,y)\in X} \frac{y}{|Y|}$, we obtain the following equations [55] for simple linear regression, which can be expressed in SQL (see Listing 3.8):

$$a = \frac{\sum\limits_{(x,y)\in X} (x - \hat{x})(y - \hat{y})}{\sum\limits_{(x,y)\in X} (x - \hat{x})^2},$$
$$b = \hat{y} - a\hat{x}.$$

```
with means as (select avg(x) as mean_x, avg(y) as mean_y from data)
select a, mean_y - a * mean_x as b
from ( select sum((x - mean_x) * (y - mean_y)) / sum((x - mean_x)^2) as a
       from data, means), means
```

LISTING 3.8: Simple linear regression in SQL.

To handle the increased number of attributes involved in multiple linear regression, we can use matrix operations as introduced in SQL. First, we aggregate n tuples with m attributes to a $n \times (m + 1)$ matrix (attributes and an added constant 1 to incorporate the weight b). Then, we gain a linear function $X \cdot \vec{w} \approx \vec{y}$ with weights $\vec{w} = (b \| \vec{a})$. Assuming we can invert the matrix

$$X' = \begin{pmatrix} 1 & x_{1,1} & \dots & x_{1,m} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n,1} & \dots & x_{n,m} \end{pmatrix} \in \mathbb{R}^{n\times(m+1)},$$

then we obtain the weights that minimise the loss function from the following equation [80], which can be expressed in our extended SQL (see Listing 3.9):

$$\vec{w} = (X'^T X')^{-1} X'^T \vec{y}.$$

```
select (array_inverse(array_transpose(x)*x))*(array_transpose(x)*y)
from (select array_agg(x) x from (select array[1,x_1,x_2] as x from data) sx) tx,
     (select array_agg(y) y from (select array[y] y from data) sy) ty;
```

LISTING 3.9: Multiple linear regression in SQL using matrix operations.

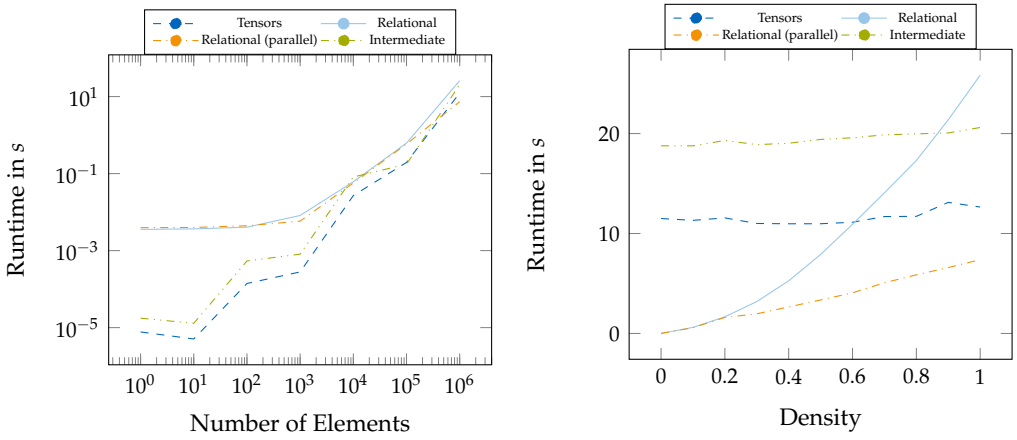
3.7 Evaluation

This section evaluates gram matrix computation on different matrix representations within HyPer and compares HyPer's performance to other systems when computing the optimal weights for linear regression. The experiments were run multi-threaded on a 20-core Ubuntu 17.04 machine (Intel Xeon E5-2660 v2 CPU) with hyper-threading, running at 2.20 GHz with 256 GB DDR4 RAM.

3.7.1 Gram Matrix Computation

To calculate the gram matrix, we vary the number of elements (by multiplying each dimension by a factor of 10 in turn) or the density (by setting elements to `null`, which are not considered by the relational representation). We ran our benchmark by filling the tensor with up to 10^6 of uniformly distributed random numbers (`rand()` from `cstdlib`).

In our tests, the implemented tensor data type outperformed both the relational representation and the intermediate format, independently of its size (see Figure 3.2a). According to [44], their results asserting the importance of a tensor data type within database systems are also valid for main-memory database systems. However, the more elements a tensor has, the more the runtime of the relational representation approximates the runtime of the built-in tensor data type. As expected, the runtime of the intermediate format lies between the runtimes of the other two implementations. In single-threaded execution, the runtime of the relational tensor representation increases with density, while the runtime of the dense tensor data type is constant (see Figure 3.2b). Nevertheless, dense tensor data types are better suited for computations when dealing with tensors with a density of more than 50 %. Only with multi-threading, the relational representation is faster due to the parallelism of the underlying database system. The intermediate format also needs constant time but is slower than the pure tensor data type. In summary, tensor data types are still needed in modern database systems, but the data types must be adapted for use with sparse data if they are to compete with the relational representation.



(A) Tensors: elements.

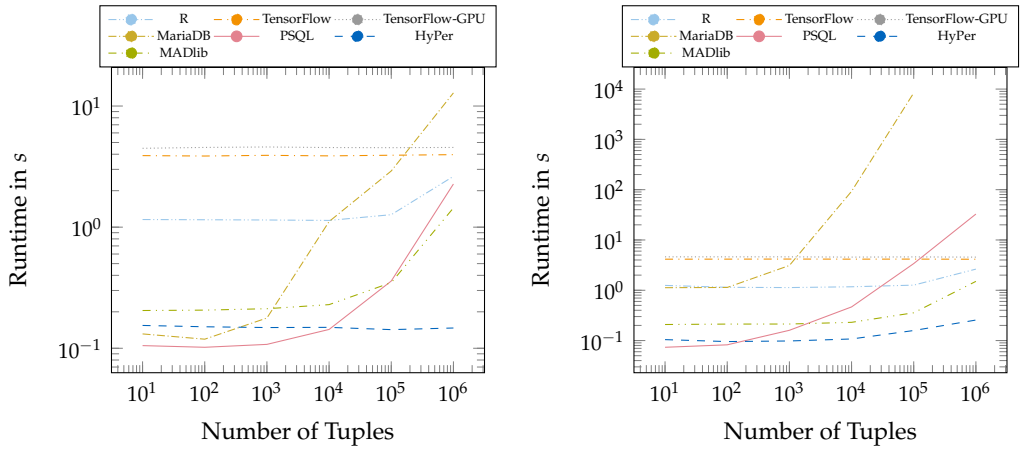
(B) Tensors: density.

FIGURE 3.2: Runtime for tensor computations: evaluation of the gram matrix computation when varying the number of elements or the density.

3.7.2 Linear Regression Using Equation Systems

The Chicago taxi rides dataset[1] was used with simple linear regression to predict the fare based on the trip distance and with multiple linear regression to also consider the ride time. As a baseline, we considered two other database systems, namely MariaDB 10.1.30, PostgreSQL 9.6.8 with the MADlib v1.13 extension, as well as two dedicated tools, namely TensorFlow 1.3.0 (with GPU support enabled and disabled) and R 3.4.2.

[1] https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew



(A) Simple linear regression.

(B) Multiple linear regression.

FIGURE 3.3: Runtime for solving (a) simple or (b) multiple linear regression by equation systems.

Simple linear regression was solved using common SQL-92 queries in HyPer and the competitor database systems. In TensorFlow, we used tensor algebra for solving equation systems for linear regression. Heyenbrock [55] created the matrix operations needed to solve multiple linear regression in PostgreSQL as PL/pgSQL function and in MariaDB as procedure[2], while we implemented them in HyPer as C++ functions and R already supports matrix algebra. The MADlib extension for PostgreSQL provides a predefined function for solving equation systems for linear regression that we used.

As Figure 3.3 shows, our in-memory database was able to carry out both linear regression tasks in less than 0.3 seconds, significantly outperforming all the baseline methods. For small numbers of tuples, even the other classical database systems performed substantially better than TensorFlow, which required at least three seconds for every task.

3.8 Conclusion

This chapter has presented algebraic operations on HyPer's and Umbra's array data type, which is similar to the one in PostgreSQL. Linear algebra operations like addition, subtraction and (scalar/elementwise) multiplication allowed this data type to simulate matrices and tensors. These extensions allowed us to solve linear regression numerically. Using gram matrix computation, we demonstrated the suitability of different matrix representations for linear algebra within a database system. For example, Chapter 8 uses a relational array representation within the database system Umbra as the basis for an array query language. As we will present in the following chapters, the extended array data type can be used for data analysis (see Chapter 4), for example, to represent sets, and machine learning (see Chapter 5, 7), i.e., to simulate a weight matrix for a neural network. As a regular data type, like for a numerical one, its operations can be used within regular SQL expressions, including SQL lambda functions.

[2] https://gitlab.db.in.tum.de/MaxEmanuel/regression_in_sql

Chapter 4

SQL for Data Analysis: HyPerScript and Recursive Tables

This chapter is based on a translation of [122].

The performance increase of database servers through modern hardware allows database systems to be used for more than pure data management tasks. In order for database servers to take over more application logic, database users should be able to develop algorithms on their own. But to extend database systems for arbitrary algorithms without implementing an operator in each case, a domain-specific language is needed that provides procedural constructs with embedded SQL. A procedural scripting language combined with the declarative query language SQL enables user-defined functions (UDF) in a more compact representation than purely imperative languages. The platform independence and reusability of SQL increases the incentive to execute complex algorithms already in the database system.

A scripting language, which allows moving computation into the database system, reduces data transfer and can be integrated within the database system's query optimiser. As stored procedures, *UDF*s are part of the query plan and thus part of the holistic query optimisation. Based on SQL, such a procedural declarative language enables data analysts to program independently of the surrounding system.

In order to identify the functional scope of such a scripting language, it is relevant how algorithms can be abstracted and what kind of building blocks are essential.

This chapter identifies necessary building blocks by implementing selected data mining algorithms (Apriori, DBSCAN, k-Means, PageRank) in *HyPerScript*, the scripting language of the main-memory database system HyPer [70], *PL/pgSQL*, the one of the disk-based database system *PostgreSQL*, and recursive tables. This chapter's contributions are:

- The presentation of *HyPerScript* as an extension of a main-memory database system allowing its users to formulate procedural statements in SQL.
- The description of a business application in the form of a TPC-C query in *HyPerScript*. This is the primary use case of this scripting language.
- The presentation of necessary building blocks within a scripting language for database systems to formulate arbitrary algorithms; demonstrated with the algorithms PageRank, k-Means, DBSCAN and Apriori, which already exist as table operators in *HyPer*.
- An evaluation of *HyPerScript* functions compared to implemented operators, recursive tables and *PL/pgSQL*.

The remainder of this chapter is structured as follows: After an introduction to *HyPerScript*, the implemented algorithms are presented. The evaluation compares the performance of this scripting language to recursive tables, *PL/pgSQL* and the operators integrated in *HyPer*.

```
Statement     = (( VarDeclaration  | VarDefinition  | TableDeclaration
                 | SelectStatement | EmbeddedSQL    | ReturnStatement
                 | IfStatement     | WhileStatement | ControlStatement | FunctionCall ) ";" )*;

VarDeclaration   = "var" NAME Type "=" Expression ;
VarDefinition    = NAME "=" Expression ;
TableDeclaration = "table" NAME "(" NAME Type ("," NAME Type)* ")";
SelectStatement  = SQLSelect ("{" Statement "}") ("else" "{" Statement "}" ) ;
EmbeddedSQL      = SQLInsert | SQLUpdate | SQLDelete | CSVCopy ;
ReturnStatement  = "return" NAME | "rollback" ;
IfStatement      = "if" "(" Expression ")" "{" Statement "}" "else" "{" Statement "}" ;
ControlStatement = "break" | "continue" ;
WhileStatement   = "while" "(" Expression ")" "{" Statement "}" ;
FunctionCall     = NAME "(" Expression ("," Expression )* ")";
```

LISTING 4.1: Language definition of *HyPerScript*: `SQL{Select,Insert,Update,Delete}` correspond to SQL:2003 commands, `Type` to SQL data types, `Expression` to SQL expressions. Besides expressions of procedural languages, `EmbeddedSQL` allows operating on individual tuples of the surrounding SQL query.

```
1 create or replace function insert_until(anzahl int not null) as $$
2    select index as i from sequence(0,anzahl){
3       var rand = random(100); if (rand = 13) { continue }; insert into sample(rand));
4    }
5 $$ language 'hyperscript' strict;
```

LISTING 4.2: Exemplary user-defined function with *HyPerScript*: `number` random numbers are inserted into a relation called `sample`, randomised between `0` and `100`, one number `13` should be skipped.

4.1 HyPerScript

HyPerScript [70] is the procedural language in *HyPer* [69], analogous to PL/SQL in Oracle [90], to *PL/pgSQL*[1] in *PostgreSQL* or to *SQLScript* in *SAP HANA* [12]. Designed for transactional applications in *HyPer*, *HyPerScript* allows executing TPC-C, TPC-E and TPC-H benchmarks. This section also describes its usage for data analysis algorithms.

The procedures defined in *HyPerScript* are translated into LLVM code during compilation. Listing 4.1 shows the language specification in extended Backus-Naur-Form (EBNF). HyPerScript uses the language constructs specified in the SQL:2003 standard [41] including window functions. SQL commands can be used within procedural constructs such as conditions (`if(<condition>){...}`), loops (`while<condition>{...}`) and iterations (`select index from sequence(1,10){...}`), together with the control commands `break` and `continue`. In addition, temporary tables can be created, variables declared and defined on the basis of the SQL data types (`var foo int[]='{}'`). Both can be passed as a parameter to a function call or returned afterwards.

Listing 4.2 demonstrates an exemplary application of *HyPerScript*. An interval is defined using declarative SQL, which is used here as an iteration similar to a `for` loop. The individual result tuples can be accessed within the scope and used in nested SQL commands. In this case, a relation called `sample` is filled with random values.

4.1.1 HyPerScript with Tensor Operations

The tensor operations introduced in Chapter 3 allow processing data aggregated to arrays. As an example, Listing 4.3 shows the cross-validation on a dataset and Listing 4.4 shows the binary exponentiation in *HyPerScript*. Binary exponentiation requires the identity matrix and matrix multiplication. This corresponds to the C++ implementation of the operation within the database system. The depicted simple cross-validation slices the dataset: one tuple serves as the test set, the remaining data as the training set. Linear regression (as a separate function) is applied to the training set to calculate the optimal weights. The optimal weights are used to determine the loss on the test set.

[1] `https://www.postgresql.org/docs/10/static/plpgsql.html`

```
create or replace function linearregression(x float[][], y float[][]) returns float[] as $$
   return (array_transpose(x)*x)^-1*(array_transpose(x)*y);
$$ language 'hyperscript' strict;
create or replace function cross_validate(x float[][], y float[][]) returns float as $$
   var error=0; var n=array_length(x,2); var m=array_length(x,1);
   select index i from sequence(2,n-1){
      var weights_o=linearregression(array_slice(x,1,i-1,1,m)||array_slice(x,i+1,n,1,m),
                          array_slice(y,1,i-1,1,n)||array_slice(y,i+1,n,1,1));
      error=error+((array_slice(x,i,i,1,m)*weights_o)[1][1]-y[i][1])^2;
   }
   error=error/(n-2); return error;
$$ language 'hyperscript' strict;
```

LISTING 4.3: Simple cross-validation in *HyPerScript*: Using `array_slice()` and concatenation, every iteration removes one row of the dataset for use as test data. `array_slice()` expects an array as first argument and the start and end indices for each dimension.

```
create or replace function pow(a_in float[][], e_in int) returns float[] as $$
   var a=a_in; var e=e_in; if( e<0 ) { e=e*-1; a=array_transpose(a); }
   var mask = 1<<63; var result = array_identity(array_ndims(a));
   while(mask>0){ result=result*result; if( e&mask>0 ){ result=result*a; } mask=mask>>1; }
   return result;
$$ language 'hyperscript' strict;
```

LISTING 4.4: Binary exponentiation `pow()` of tensors implemented in *HyPerScript*: a tensor and an exponent are expected as input, a loop with multiplications calculates the exponentiation.

4.1.2 HyPerScript for Business Transactional Applications

HyPerScript allows business transactional applications to be executed via the SQL interface. Examples are the TPC-C, TPC-E and TPC-H benchmarks of the Transaction Processing Performance Council [2]. For example, the TPC-C benchmark models a trading company with incoming orders (`newOrderPosition`) from customers and measures how many write transactions a database system can process per time unit.

```
create type newOrderPosition as (line_number int not null,supware int not null,itemid int not
    null, qty int not null);
create function newOrder (w_id int not null, d_id smallint not null, c_id int not null,
    positions setof newOrderPosition not null, datetime timestamp not null) as $$
  select w_tax from warehouse w where w.w_id=w_id;
  select c_discount from customer c where c_w_id=w_id and c_d_id=d_id and c.c_id=c_id;
  select d_next_o_id as o_id,d_tax from district d where d_w_id=w_id and d.d_id=d_id;
  update district set d_next_o_id=o_id+1 where d_w_id=w_id and district.d_id=d_id;
  select count(*) as cnt from positions;
  select case when count(*)=0 then 1 else 0 end as all_local from positions where supware<>w_id;
  insert into "order" values (o_id,d_id,w_id,c_id,datetime,null,cnt,all_local);
  insert into neworder values (o_id,d_id,w_id);
  update stock
  set s_quantity=case when s_quantity>=qty+10 then s_quantity-qty else s_quantity+91-qty end,
      s_remote_cnt=s_remote_cnt+case when supware<>w_id then 1 else 0 end,
      s_order_cnt=s_order_cnt+1, s_ytd=s_ytd+qty
  from positions where s_w_id=supware and s_i_id=itemid;
  insert into orderline
         select o_id,d_id,w_id,line_number,itemid,supware,null,qty,
                qty*i_price*(1.0+w_tax+d_tax)*(1.0-c_discount),
                case d_id when 1 then s_dist_01 when 2 then s_dist_02 when 3 then s_dist_03 when
                      4 then s_dist_04 when 5 then s_dist_05 when 6 then s_dist_06 when 7 then
                     s_dist_07 when 8 then s_dist_08 when 9 then s_dist_09 when 10 then
                     s_dist_10 end
         from positions, item, stock
         where itemid=i_id
         and s_w_id=supware and s_i_id=itemid
  returning count(*) as inserted;
  if (inserted<cnt) rollback;
$$ language 'hyperscript' strict;
```

LISTING 4.5: *HyPerScript* function `newOrder()` [68, 70] of the TPC-C benchmark: The function takes a set of order items `positions` for a warehouse `w_id` of a district `d_id` from a customer `c_id`.

Listing 4.5 shows the corresponding procedure in *HyPerScript* taken from [68, 70]. The procedure takes a new order with several items (`setof`) for a warehouse (`w_id`) from one customer (`c_id`) in a district (`d_id`). First, this order is inserted into the database (line `9/10`). Then the procedure updates the `stock` and inserts the order item together with the calculated price in `orderline` (line `16-23`).

[2] http://www.tpc.org

HyPerScript allows storing the result of an SQL query temporarily for later reuse. For example, the variables `w_tax` and `c_discount` (line `3/4`) cache the warehouse's tax rate and the customer discount, which are reused when inserted into the relation `orderline`.

The example checks the atomicity and consistency for transactions, which is guaranteed by the use of a database system. At the end (line `24`), the procedure checks whether the orders have been successfully inserted into `orderline` (the number of inserted orders is valid) and resets the transaction otherwise (`rollback`). In addition, the example illustrates the compact notation: The *HyPerScript* procedures needs less than 40 lines of code, while a C++ program for the same function requires considerably more lines[3].

4.2 SQL for Data Analysis

Like other scripting languages, *HyPerScript* allows running arbitrary algorithms within the database system besides the intended use for performance analysis. Although SQL is already Turing-complete when supporting recursive tables[4], a scripting language can provide a compact and understandable way to formulate algorithms. This section presents a set of data analysis algorithms written in recursive SQL, *HyPerScript* and *PL/pgSQL*: algorithms for association rule analysis (Apriori), clustering (k-Means and DBSCAN) and graph metrics (PageRank).

Each implementation of an algorithm in a scripting language follows a predefined structure: For each algorithm in a scripting language, a schema is defined as a separate namespace. A call to `<schema name>.run()` runs the algorithm. Afterwards, the result has been inserted into a relation of the same namespace. Using *HyPerScript* and *PL/pgSQL*, we embed SQL-92 queries within procedural statements for iterations. We then transform each iteration into a recursion using recursive tables. The selected algorithms are available as parallelised operators within HyPer, which serve for comparison [102, 134].

4.2.1 PageRank

PageRank [20] is a graph mining algorithm designed to determine the importance of web pages. Each web page is called a node $n \in N$; a link directing to another web page is called an edge $(s,d) \in N \times N$. Initially, each node receives the same PageRank value pr_0 (Equation 4.1). In each iteration, each node s distributes its own value $pr_i(s)$ equally to all outgoing edges $(s,d) \in E$. The new PageRank value $pr_{i+1}(n)$ of a node n is the sum of the values of all incoming edges $(s,n) \in E$, possibly damped by a factor α (Equation 4.2):

$$pr_0(n) := \frac{1}{|N|}, \tag{4.1}$$

$$pr_{i+1}(n) := \alpha \cdot \sum_{(s,n)\in E} \frac{pr_i(s)}{|\{d \mid (s,d) \in E\}|} + \frac{1-\alpha}{|N|}. \tag{4.2}$$

```
1 create table prscript.edges (src int, dst int);
2 -- native operator:
3 select * from pagerank((table prscript.edges),0);
4 -- HyPerScript:
5 select prscript.run(0); select * from prscript.pagerank;
```

LISTING 4.6: Calling PageRank in HyPer: Either as an operator, which needs the edges and the damping factor as input, or as a script function.

Both equations can be described in SQL with aggregations, whereas for the iteration we need either loops of a scripting language like HyPerScript (see Listing 4.7) or a recursive table (see Listing 4.8). We expect one relation containing the edges (see Listing 4.6). The base case (line `2/3`, Listing 4.8) computes the initial PageRank value pr_0. The recursive step first divides each node's PageRank value by the number of outgoing edges (line `7`) and assigns this fraction to the destination node `dst`. It then sums up the fractions for each destination node (line `5-11`).

[3] https://github.com/evanj/tpccbench/blob/master/tpcctables.cc
[4] https://wiki.postgresql.org/index.php?title=Cyclic_Tag_System

```
create or replace function prscript.computePR(a float not null) returns int not null as $$
 table pr_tmp(node int, edges int); -- new PageRank values
 insert into pr_tmp (
  select dst, (1-a)*(cast((select count(*) from prscript.pagerank) as float))+a*sum(b)
  from (
   select e.dst, p.Pagerank/(select count(*) from prscript.edges x where x.src=e.src) as b
   from prscript.edges e, prscript.pagerank p where e.src=p.Node) i group by dst);
 select count(*) as c_count from (select * from prscript.pagerank except select * from pr_tmp);
 truncate prscript.pagerank; -- update PageRank values
 insert into prscript.pagerank (select * from pr_tmp);
 return c_count; -- return number of updated values
$$ language 'hyperscript' strict;

create or replace function prscript.run(a float not null, maxIter int not null) as $$
   select index as i from sequence(0,maxIter) {
      select prscript.computePR(a) as c_count;
      if(c_count = 0){ break; }
   }
$$ language 'hyperscript' strict;
```

LISTING 4.7: PageRank in HyPerScript: *computePR* calculates the PageRank values, `a` is the damping factor, `run` manages the iterations.

```
with recursive pagerank (iter,node,pr) as (
 select 0, e.dst, 1::float/(select count(distinct dst) from prscript.edges)
 from prscript.edges e group by e.dst
union all
 select iter+1,dst,0.1*((1::float/(select count(distinct dst) from prscript.edges)))+0.9*sum(b)
 from (
  select iter, e.dst, p.pr/(select count (*) from prscript.edges x where x.src=e.src) as b
  from prscript.edges e, pagerank p
  where e.src=p.Node and iter < 100
 ) i
 group by dst, iter
)
select * from pagerank where iter=100;
```

LISTING 4.8: PageRank in SQL with a recursive table `pagerank` and $\alpha = 0.9$.

4.2.2 Clustering

Clustering is an important area of data analysis that groups similar tuples. We consider k-Means and DBSCAN [112], as both algorithms are integrated into HyPer as operators.

```
create table kmeansscript.points (id varchar(8), x float, y float);
create table dbscanscript.points (id varchar(8), x float, y float);
-- native operator:
select * from kmeans((select x,y from kmeansscript.points),
                     (select x,y from kmeansscript.points LIMIT 5));
select * from dbscan((select x,y from dbscanscript.points),20,2);
-- HyPerScript:
select kmeansscript.run(5); select dbscanscript.run(20,2);
```

LISTING 4.9: Calling the clustering algorithms in HyPer: The k-Means operator takes the data points and the initial centres ($k = 5$). The DBSCAN operator expects the data points, ϵ and the minimum number of points per cluster as parameters. The HyPerScript functions require either the parameter k (k-Means) or ϵ and the minimum number of points per cluster (DBSCAN).

k-Means assigns n-dimensional points $x \in P \subset \mathbf{R}^n$ to k clusters C with k points forming the initial centres $C_0 \subset P, |C_0| = k$. A point belongs to the closest located cluster $c \in C$ based on a metric like Euclidean distance ($||c - x||_2$, Equation 4.3 returns all points of a cluster). In each iteration, the new centre is computed as the average of all points (Equation 4.4):

$$cluster(c) = \{x | x \in P : \nexists d \in C : d \neq c \wedge ||d - x||_2 < ||c - x||_2\}, \tag{4.3}$$

$$c_{i+1} = \sum_{x \in cluster(c_i)} \frac{x}{|cluster(c_i)|}. \tag{4.4}$$

The HyPerScript implementation for computing the centres of k-Means (see Listing 4.10) is based on a window function [82] that calculates a ranking of the closest centres per point (line `13/14`). The mean values of the assigned points form the new centres (line `12-15`). Using a recursive table (see Listing 4.11), k tuples are initially selected as centres (line `2`), whose coordinates get updated in each iteration (line `4-8`).

```
create or replace function kmeansscript.run(k int not null) as $$
   truncate kmeansscript.clusters;
   insert into kmeansscript.clusters(select id,x,y,0 from (select *,rank() over (order by id)
        from kmeansscript.points) where rank <= k);
   while (true){
      select kmeansscript.computeCenters() as c_count;
      if(c_count = 0){ break; }
   }
$$ language 'hyperscript' strict;

create or replace function kmeansscript.computeCenters() returns int not null as $$
   table clusters_tmp(cid int, x float, y float, count int);
   insert into clusters_tmp (select cid, avg(px), avg(py), count(*) from (
       select cid, p.x as px, p.y as py, rank() OVER ( partition by p.id
         order by (p.x-c.x)*(p.x-c.x)+(p.y-c.y)*(p.y-c.y) asc, (c.x*c.x+c.y*c.y) asc)
       from kmeansscript.points p, kmeansscript.clusters c ) x where x.rank=1 group by cid);
   select count(*) as c_count from (select * from kmeansscript.clusters except select * from
        clusters_tmp);
   truncate kmeansscript.clusters; -- delete old centres
   insert into kmeansscript.clusters (select * from clusters_tmp);
   return c_count; -- return number of updated centres
$$ language 'hyperscript' strict;
```

LISTING 4.10: k-Means in HyPerScript: `computeCenters()` computes the centres on the two-dimensional data of the relation `points`, which is called by `run()` until the centres are found.

```
with recursive clusters (iter, cid, x, y) as (
  (select 0,id, x, y from kmeansscript.points limit 5)
union all
  select iter+1,cid, avg(px), avg(py) from (
    select iter, cid, p.x as px, p.y as py, rank() over (partition by p.id
      order by (p.x-c.x)*(p.x-c.x)+(p.y-c.y)*(p.y-c.y) asc, (c.x*c.x+c.y*c.y) asc)
    from kmeansscript.points p, clusters c) x
  where x.rank=1 and iter<100 group by cid, iter
)
select * from clusters where iter=100;
```

LISTING 4.11: k-Means in SQL with a recursive table `clusters`: k=5, 100 iterations.

DBSCAN clusters points depending on a parameter ϵ, that describes the maximal distance between two points, and the minimal number of points per cluster $minPoints > 1$, declared as noise otherwise. For every point within a cluster $x \in G \subset P \subset \mathbf{R}^n$, another point $y \in G$ exists, whose distance to x is less than ϵ:

$$\forall x \in G : \exists y \in G : x \neq y \wedge ||x - y||_2 < \epsilon. \tag{4.5}$$

The implementation of DBSCAN in *HyPerScript* (see Listing 4.12) creates a cluster around each unclustered point iteratively: First, an unclustered point is taken (line 11) to form a new cluster, then all surrounding points are added to the cluster (line 13–21).

```
create or replace function dbscanscript.run(eps float, minPoints int) as $$
   select count(*) as maxiter from dbscanscript.points;
   select index from sequence (1,maxiter){
      select dbscanscript.insertCluster(index,eps,minPoints,maxiter) as cond;
      if(cond=0){ break; }
   }
$$ language 'hyperscript' strict;

create or replace function dbscanscript.insertCluster(clusterid int not null, eps float not null
     , minPoints int not null, maxiter int not null) returns int not null as $$
 table pointstmp (id int, x float, y float);
 insert into pointsTmp(select * from dbscanscript.points except (select id,x,y from dbscanscript
      .pointsclustered) LIMIT 1);
 select count(*) as ret from pointsTmp;
 select index from sequence(1,maxiter){
  select count(*) as newc from pointsTmp c, dbscanscript.points p
   where (p.x-c.x)^2+(p.y-c.y)^2<eps^2 and c.id<>p.id;
  insert into pointsTmp(
   select * from (select * from dbscanscript.points except select * from pointsTmp) c
   where exists (select * from pointsTmp p where (p.x-c.x)^2+(p.y-c.y)^2<eps^2 and c.id<>p.id));
  if(newc=0){ break; }
  ret=ret+newc;
 }
 if(ret < minPoints) { clusterid=NULL; }
 insert into dbscanscript.pointsclustered(select *,clusterid, false from pointsTmp);
 return ret;
$$ language 'hyperscript' strict;
```

LISTING 4.12: DBSCAN in HyPerScript: `insertCluster()` creates a cluster around an unclustered point and returns the cluster size.

When the database system supports aggregate functions within recursive tables, the clusters can be expanded recursively (see Listing 4.13): First, each point forms its own cluster (line 2). Then, clusters that are less than ϵ away are merged (line 4–7).

```
1 with recursive dbscan(iter,id,x,y,clusterid,noise) as (
2   select 0,id,x,y,id,true from dbscanscript.points
3 union all
4   select iter+1,p.id,p.x,p.y, min(c.clusterid), count(*) < 3
5   from dbscanscript.points p, dbscan c
6   where iter<10 and (p.x-c.x)^2+(p.y-c.y)^2<1.5^2
7   group by iter,p.id,p.x,p.y
8 ) select * from dbscan where iter=10;
```

LISTING 4.13: DBSCAN in SQL with a recursive table `dbscan`: 10 iterations, $\epsilon = 1.5$, $minPoints = 3$.

For comparison, we use the operators k-Means and DBSCAN implemented in HyPer. As a special feature, both operators are written directly in LLVM code, are compiled directly into the query and avoid expensive function calls. Both operators are pipeline breakers, but the centre calculation for k-Means does not require materialisation of the data points as it copies the underlying operator tree per iteration.

4.2.3 Association Rule Analysis

The Apriori algorithm [3], introduced in 1993, is the best-known representative in the field of association rule analysis. It is based on shopping cart data stored as tuples out of transaction number (*tid*) and item (sales: $\{[tid, item]\}$). First, it selects frequent item sets that occur with a minimum relative frequency, *support*, of at least s_0 in all shopping carts, which are used to create association rules. In each iteration, the item sets grow by one element, starting with the one-element set. Here, the number of iterations and item sets to be checked is limited by the *apriori principle*. The principle states that an item set whose subsets do not occur frequently cannot be a frequent item set. Listing 4.14 shows the call of the integrated operator as well as of the script function with a minimum support of 10 %.

```
1 create table aprioriscript.sales (tid int, item int);
2 -- native operator:
3 select * from apriori((table aprioriscript.sales),0.1,1);
4 -- HyPerScript
5 select aprioriscript.findFI(10); select * from aprioriscript.frequentitemsets;
```

LISTING 4.14: Call of the Apriori algorithm as a database operator and as a stored procedure.

The implementation in *HyPerScript* (see Listing 4.15) is based on an array representation for frequent item sets, which are iteratively extended with recursive SQL. Here, arrays are used as sets and expanded by one element in each iteration. Then the support of each item set is counted. For this purpose, each item set is compared with each shopping cart using the set operator `tuple <@ shopping cart` (line 13).

```
1  create or replace function aprioriscript.findFrequentItemsets(minsupp int not null) as $$
2     with recursive transactions (tid, bucket) as ( -- create one array per shopping cart
3        select tid, array_agg(item) from aprioriscript.sales group by tid),
4     -- frequent item sets of size 1
5     sales_supp as (select item from aprioriscript.sales group by item having count(*)>= minsupp),
6     frequentitemsets as ( -- frequent item sets with support >= minsupp
7        (select distinct array[p.item]::int[] as items from sales_supp p) -- with one element
8       union all ( -- extend item sets recursively by one element
9         select distinct array_append(t.items,p.item::int)::int[]
10        from frequentitemsets t, sales_supp p
11        where minsupp <= ( -- count support
12           select count(*) from transactions t2
13           where array_append(t.items,p.item::int)::int[] <@ ( t2.bucket )
14        ) and t.items[(select count(*) from unnest(t.items))]<p.item
15    ))
16    insert into aprioriscript.frequentitemsets (select * from frequentitemsets);
17 $$ language 'hyperscript' strict;
```

LISTING 4.15: Determining the frequent item sets for the Apriori algorithm: The function expects the minimum support s_0 as a parameter and calculates a recursively growing relation with frequent item sets, starting with the one-element item sets (each item as an array with one element).

The operator implemented in *HyPer* is based on storing the elements in a prefix tree that grows with each iteration. Special features of the implementation in *HyPer* are the parallelism per iteration step and the handling of duplicates. Thus, the association rules consider the support of identical elements within a shopping cart.

4.3 Evaluation

The evaluation compares the performance of procedures created with *HyPerScript* to table operators of HyPer and MADlib, recursive tables in HyPer, PostgreSQL and Umbra, and *PL/pgSQL* procedures (PostgreSQL 12.6 with MADlib 1.17.0 extension). All experiments were measured on a Ubuntu 20.04 LTS machine with six Intel Core i7-3930K CPUs running at 3.20 GHz and 64 GB DDR4 RAM.

For the Apriori algorithm, 100 different items and 1000 shopping carts were synthetically generated. The number of items per shopping cart varied between 0 and 10. For clustering, we generated 10^6 points whose x- and as y-coordinates were equally distributed in the interval $[0, 10^6]$. The PageRank value was computed for 10^5 nodes with the same number of edges. All experiments were repeated three times and the median was taken for the measurements.

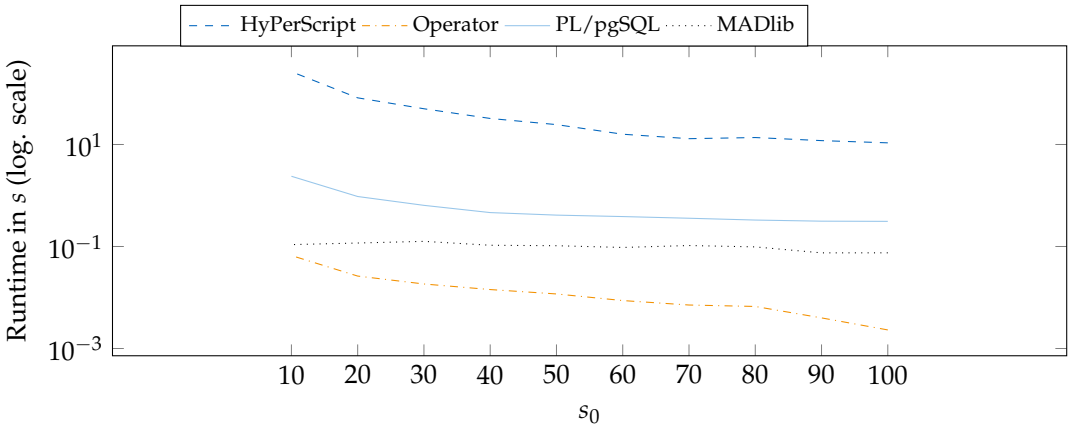
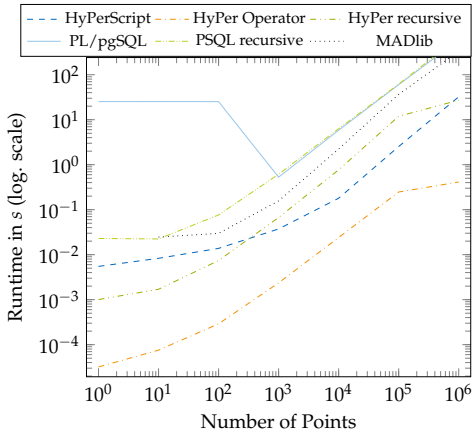
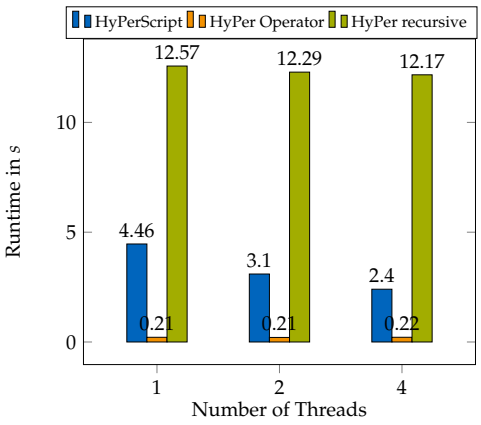


FIGURE 4.1: Runtime for association rule analysis with Apriori with twelve threads, depending on the minimum support s_0 as a parameter of Apriori: The larger s_0, the lower the number of frequent item sets and thus the lower the runtime.

For the Apriori algorithm, we varied the minimum support (the larger, the less frequent item sets exist, see Figure 4.1). With increasing minimum support, the runtime decreases as less frequent item sets exist. Although the HyPer operator performs the best, the implementation in *HyPerScript* is slower than its counterpart in *PL/pgSQL*. This is caused by an implementation of the array set operator within HyPer that unnests the array internally.



(A) k-Means: Input size.



(B) k-Means: Scalability.

FIGURE 4.2: Runtime for k-means (five centres, 100 iterations): (a) Runtime depending on the input size with constant twelve threads and (b) depending on the available threads for 10^5 points.

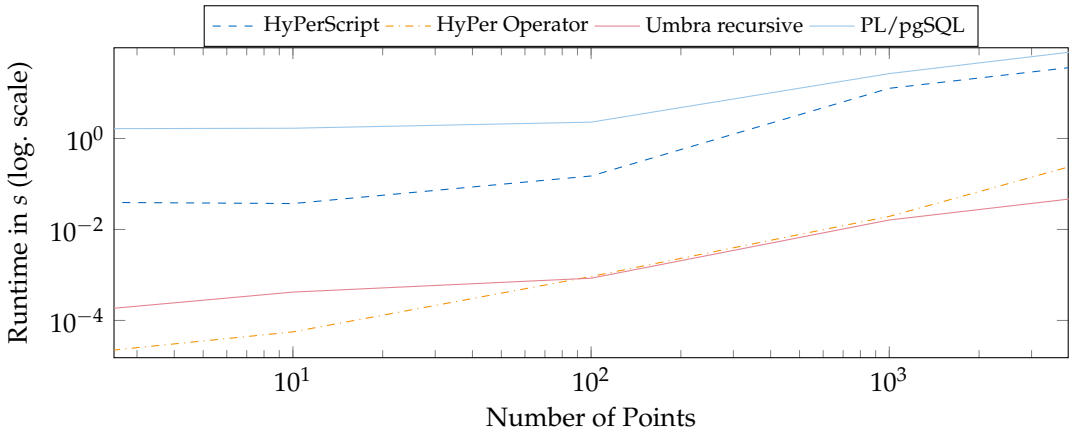
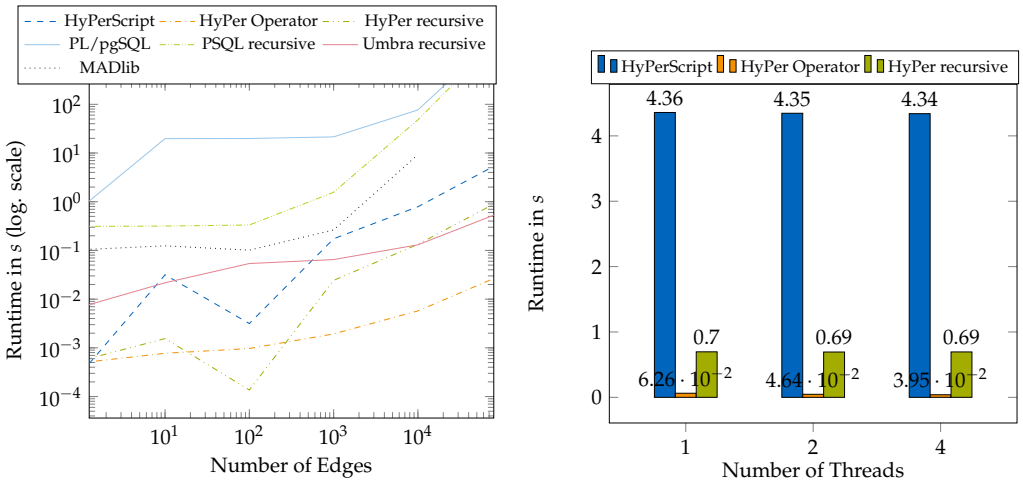


FIGURE 4.3: Runtime for DBSCAN with $\epsilon = 20$, `minPts` $= 2$ and 100 iterations: Runtime depending on the input size with constant twelve threads.

The runtimes of the clustering algorithms grow linearly with the input size (see Figure 4.2a, 4.3). Although the integrated operators perform the best, the k-Means implementation within *HyPerScript* and using a recursive table (in HyPer, as Umbra's support for window functions was in development at that time) show comparable performance, both outperform the computations in PostgreSQL. The recursive computation of DBSCAN in Umbra (as the other database systems do not support `min` as aggregate function inside a recursive table) was as fast as the implemented operator. The k-Means algorithm in *HyPerScript* computes 30 % faster with each additional core (see Figure 4.2b), as the underlying database system executes the SQL queries in parallel, whereas the k-Means operator is explicitly parallelised. Neither DBSCAN as an operator nor as a stored procedure support scaling. The SQL queries used in the *HyPerScript* procedure scale poorly because only one tuple is initialised as a new cluster per (non-parallelised) iteration.



(A) PageRank: Number of edges.

(B) PageRank: Scalability.

FIGURE 4.4: Runtime for 100 iterations for calculating the PageRank value with (a) increasing number of edges (twelve threads) or (b) threads (10^5 edges).

All implementations of PageRank in HyPer outperform their counterparts in PostgreSQL (see Figure 4.4). The implementations within a scripting language perform slightly worse than the corresponding query using a recursive table in PostgreSQL and HyPer respectively. With few edges, the additional overhead of the integrated operator in HyPer becomes apparent, since it creates a dictionary for the nodes and stores edges in a sparse matrix as Compressed-Sparse-Row (CSR). With an increasing number of edges, the additional effort

for the dictionary and the CSR data structure is amortised, so that the operator calculates the PageRank value faster than the script function.

4.4 Conclusion

The aim of this chapter was to explore the potential of an SQL scripting language for data analysis. We have shown that user-defined functions in a declarative language extended by procedural expressions allow a compact notation, implicitly parallelise and are executable within the database system. This eliminates expensive ETL costs and allows the database system to exploit the performance of modern hardware adequately.

The evaluation showed the performance of algorithms written in a scripting language: faster within an in-memory database system than within traditional database systems and its extensions, but slower than HyPer's highly tuned operators. The implemented procedures allowed parallelism only if the underlying SQL queries scaled accordingly, as could be seen with k-Means clustering. In summary, *HyPerScript* makes it possible to develop prototypes without extending the relational algebra of the database system. When using recursive tables, the support of aggregate and window functions is necessary to simplify the development of data analysis algorithms.

Chapter 5

In-Database Machine Learning

Parts of this chapter have been published in [121, 124].

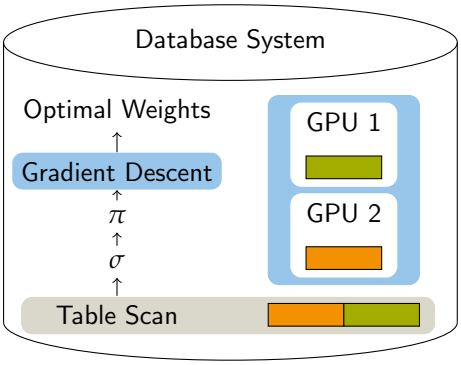


FIGURE 5.1: Architecture for in-database machine learning: a gradient descent operator, embedded in a query plan, expects training data and returns the optimal weights. Inside a dedicated operator, we are later able to off-load to GPU.

Typically, steps of machine learning pipelines—that consist of data preprocessing, model training/validation and finally its deployment on unlabelled data—are embedded in Python scripts that call up specialised tools such as NumPy, TensorFlow, Theano or Pytorch. Hereby, especially tensor operations and model training are co-processed on graphical processing units (GPUs) or tensor processing units (TPUs) developed for this purpose.

Integrating machine learning pipelines into database systems is a promising approach for data-driven applications. Even though specialised tools will outperform general-purpose solutions, we argue that an integration in database systems will simplify data provenance and its lineage, and allows complex queries as input. So far, machine learning pipelines inside of database queries are assembled from user-defined functions and operators of an extended relational algebra. This brings the model close to the data source with SQL as the only query language. As modern HTAP main-memory database systems such as SAP HANA [96], HyPer [69] and Umbra [100] are designed for transactional and analytical workload, this allows the latest database state to be queried. But for continuous machine learning based on the latest tuples, only stand-alone solutions exist [36] whose pipelines retrain weights for a model partially when new input data is available.

We argue that SQL is sufficient to formulate a complete machine learning pipeline. The database system Umbra is a computational database engine that offers—in addition to standardised SQL:2003 features—a matrix data type, a data sampling operator [15] and continuous views [138]. A continuous view updates precomputed aggregates on incoming input. This kind of view—combined with sampling—can be used as source to train and retrain a model partially within a recursive table.

This chapter starts by expressing gradient descent as a recursive table as well as the views needed for data preprocessing in SQL. Instead of manually deriving the gradient, we propose an operator for automatic differentiation. Based on automatic differentiation, we will proceed with a separate operator for gradient descent to later be able to off-load work to GPUs (see Figure 5.1). This chapter's contributions are

- machine learning pipelines expressed in pure SQL,
- automatic differentiation as an operator in SQL that uses a lambda function to derive the gradient and generates LLVM code,
- and the integration of gradient descent as a database operator.

In detail, we focus on data preprocessing for machine learning pipelines and recursive computation of gradient descent within a code-generating database system (Section 5.1). During code-generation, an operator for automatic differentiation compiles the gradient from a lambda expression (Section 5.2). Section 5.3 presents its extension to deal with matrix operations and its application on neural networks in SQL. Section 5.4 evaluates the performance increase through automatic differentiation and compares the performance of an operator for gradient descent to external tools.

5.1 In-Database Gradient Descent

This section first introduces mini-batch gradient descent, before describing a machine learning pipeline out of preprocessing and model training expressed in SQL.

5.1.1 Mini-Batch Gradient Descent

Given a set of tuples $(\vec{x}, y) \in X \subseteq (\mathbb{R}^m, \mathbb{R})$ with m features, a label and weights $\vec{w} \in \mathbb{R}^m$, then optimisation methods such as gradient descent try to find the best parameters $\vec{w}_\infty$ of a *model function* $m_{\vec{w}}(\vec{x})$, e.g., a linear function (Equation 5.1) that approximates the given label y. A *loss function* $l_{\vec{x},y}(\vec{w})$ measures the deviation (*residual*) between an approximated value $m_{\vec{w}}(\vec{x})$ and the given label y, for example, mean squared error (Equation 5.2):

$$m_{\vec{w}}(\vec{x}) = \sum_{i=1}^{m} x_i \cdot w_i = \vec{x}^T \cdot \vec{w} \approx y, \tag{5.1}$$

$$l_{\vec{x},y}(\vec{w}) = (m_{\vec{w}}(\vec{x}) - y)^2, \tag{5.2}$$

$$l_X(\vec{w}) = \frac{1}{|X|} \sum_{(\vec{x},y)\in X} l_{\vec{x},y}(\vec{w}) = \frac{1}{|X|} \sum_{(\vec{x},y)\in X} (m_{\vec{w}}(\vec{x}) - y)^2. \tag{5.3}$$

To minimise $l_X(\vec{w})$, gradient descent updates the weights per iteration by subtracting the loss function's gradient times the learning rate γ until the optimal weights $\vec{w}_\infty$ are approximated (Equation 5.6). *Stochastic gradient descent* takes one tuple for each step (Equation 5.4), whereas *batch gradient descent* considers all tuples per iteration and averages the loss (Equation 5.5):

$$\vec{w}_{t+1} = \vec{w}_t - \gamma \cdot \nabla l_{\vec{x},y}(\vec{w}_t), \tag{5.4}$$

$$\vec{w}_{t+1} = \vec{w}_t - \gamma \cdot \nabla l_X(\vec{w}_t) = \vec{w}_t - \gamma \cdot \frac{1}{|X|} \sum_{(\vec{x},y)\in X} \nabla l_{\vec{x},y}(\vec{w}), \tag{5.5}$$

$$\vec{w}_\infty \approx \lim_{t\to\infty} \vec{w}_t. \tag{5.6}$$

Smaller batch sizes, mini-batches, are mandatory when the entire input does not fit into memory and allows parallelism later on. Therefore, *mini-batch gradient descent* splits an input dataset X into disjoint mini-batches $X = X_0 \uplus \cdots \uplus X_o$.

Using a recursive table, gradient descent can be expressed in SQL. Listing 5.1 shows five iterations based on an exemplary loss function with two weights (Equation 5.7): First, the weights get initialised (line 2), then each iteration updates the weights (Equation 5.5, line 3) based on manually derived gradients (Equation 5.8) and $\gamma = 0.05$:

$$l_{x,y}(a,b) = (a \cdot x + b - y)^2 \tag{5.7}$$

$$\nabla l_{x,y}(a,b) = \begin{pmatrix} \partial l/\partial a \\ \partial l/\partial b \end{pmatrix} = \begin{pmatrix} 2(ax+b-y)\cdot x \\ 2(ax+b-y) \end{pmatrix}. \tag{5.8}$$

```
create table data (x float, y float); insert into data ...
with recursive gd (id, a, b) as (select 0,1::float,1::float UNION ALL
 select id+1, a-0.05*avg(2*x*(a*x+b-y)), b-0.05*avg(2*(a*x+b-y))
 from gd, data where id<5 group by id,a,b)
select * from gd order by id;
```

LISTING 5.1: Gradient descent using a recursive table (manually derived).

To avoid overfitting, one mini-batch or an explicitly given input will serve as the validation dataset for accuracy measurements. The remaining mini-batches will be used as input for every iteration of parallel and distributed mini-batch gradient descent.

To avoid underestimation of the predicted values, Derakhshan et al. [36] propose root mean squared logarithmic error (RMSLE) on the validation dataset:

$$\vec{w}_{t+1} = \vec{w}_t - \gamma \cdot \sqrt{\frac{1}{|X|} \sum_{(\vec{x},y) \in X} (log((m_{\vec{w}}(\vec{x}) + 1) - log(y+1))^2} \tag{5.9}$$

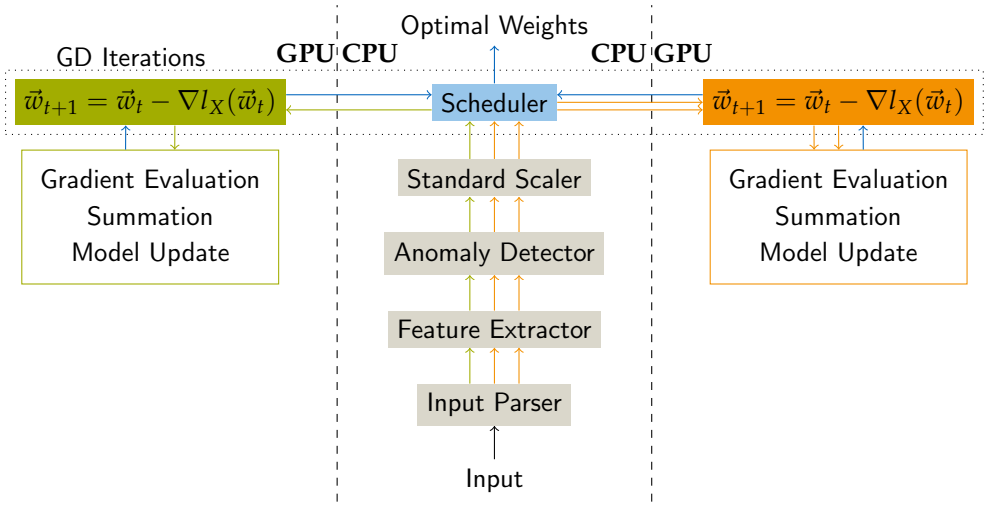


FIGURE 5.2: Components of a machine learning pipeline: chunked input will be processed independently (potentially off-loaded to GPUs). After every iteration, the weights (blue) are synchronised.

5.1.2 Machine Learning Pipeline in SQL

We argue that SQL offers all components needed for data preprocessing, and recursive tables allow gradient descent to be performed. Thus, we reimplemented the components of a machine learning pipeline (see Figure 5.2) proposed by Derakhshan et al. [36] in SQL (see Figure 5.3):

- The **Input Parser** parses input CSV files and stores the data in chunks using row-major format to allow batched processing of mini-batch gradient descent. In SQL, this corresponds to a simple table scan. In Umbra, we can also use a foreign table as input for continuous views (table `taxidata`).
- The **Feature Extractor** extracts features from data chunks, which is a simple projection in SQL. For example, day and hour are extracted from timestamps, distance metrics from given coordinates (view `processed`).
- The **Anomaly Detector** deletes tuples of a chunk on anomalies. An anomaly occurs when at least one attribute in a tuple passes over or under a predefined threshold. For anomalies, we filter for user-defined limits in a selection (view `normalised`).

- The **Standard Scaler** scales all attributes in the range $[0,1]$ to equal each attribute's impact on the model, this corresponds again to a projection and nested subqueries to extract the attribute's extrema (view `normalised`).
- The **Scheduler** manages gradient descent iterations until the weights converge. This can be either done using recursive tables or using an operator that off-loads work to GPU.

Listing 5.2 shows the resulting SQL queries using a taxi dataset as exemplary input and a linear function to predict a trip's duration based on its day, hour, distance and bearing. In this example, we perform 50 iterations of mini-batch gradient descent based on a sample size of ten tuples (`tablesample reservoir (10)`) and a learning rate of 0.001. In every iteration, we subtract the average gradient from the weights (line `7-14`), which we finally use to compute the loss (line `15/16`). As computing each partial derivative manually can be bothersome and error-prone for complex loss functions, we proceed with an operator for automatic differentiation in the next section.

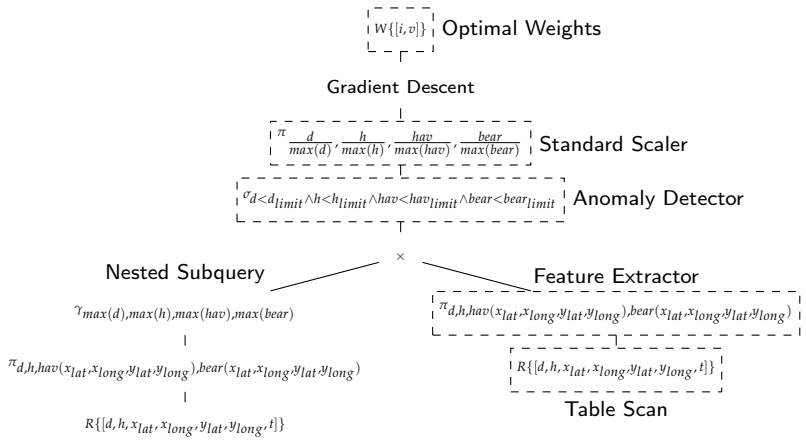


FIGURE 5.3: Operator plan inside of a database system with linear regression on the New York taxi dataset in relational algebra: a projection extracts the features as haversine (*hav*) distance or bearing (*bear*), anomalies are deleted using predefined thresholds (denoted as *limit*).

```
1  create foreign table taxidata(id int, pickup_datetime date, dropoff_datetime date, passengers
       float, pickup_longitude float, pickup_latitude float, dropoff_longitude float,
       dropoff_latitude float, duration float) server stream;
2  copy taxidata from './taxidata.csv' delimiter ',';
3  create view processed as (select hour,day,duration,ACOS(SIN(plat)*SIN(dlat)+COS(plat)*COS(dlat)*
       COS(dlong-plong))*6371000 distance, ATAN2(SIN(dlong-plong)*COS(dlat),COS(plat)*SIN(dlat)-
       SIN(plat)*COS(dlat)*COS(dlong-plong))*180/PI() bearing from (select avg(hour) as hour, avg(
       day) as day, avg(duration) as duration, avg(plat) as plat, avg(plong) as plong, avg(dlat)
       as dlat, avg(dlong) as dlong from (select cast(extractHour(dropoff_datetime) as float) as
       hour,cast(extractDay(dropoff_datetime) as day,duration, pickup_latitude/180*pi() plat,
       pickup_longitude/180*pi() plong, dropoff_latitude/180*pi() dlat, dropoff_longitude/180*pi()
        as dlong from taxidata) group by hour, day, duration, plat, plong, dlat, dlong));
4  create view normalised(hour, day, distance, bearing, duration) as (select cast(hour as float)/(
       select max(hour)+1 from processed), cast(day as float)/(select max(day) from processed),
       distance/(select max(distance) from processed where distance < 1000), (bearing+360)
       %360/360.0, duration/(select max(duration) from processed) from processed where distance <
       1000);
5  with recursive gd (id, a1, a2, a3, a4, b) as (
6  select 1, 1::float, 1::float, 1::float, 1::float, 1::float UNION ALL
7  select id+1,
8    a1-0.001*avg(2*hour*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
9    a2-0.001*avg(2*day*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
10   a3-0.001*avg(2*distance*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
11   a4-0.001*avg(2*bearing*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
12   b -0.001*avg(2*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration))
13 from gd, (select * from normalised tablesample reservoir (10))
14 where id<=50 group by id,a1,a2,a3,a4, b)
15 select id, avg(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)^2
16 from gd,normalised where id=51;
```

LISTING 5.2: Machine learning pipeline in SQL.

5.2 Database Operators for Machine Learning

This section describes the operators in Umbra and HyPer, which we created to facilitate machine learning in SQL. Modern database systems generate code for processing chunks of tuples in parallel pipelines, so we first explain code-generation before we present the operators for automatic differentiation and gradient descent.

5.2.1 Code Generation

In database systems, operators can be combined to form an operator tree with table scans as leaves. With Umbra and HyPer as the integration platform, an operator follows the concept of a code-generating database system. It achieves parallelism by starting as many pipelines as threads are available and expects each operator in a query plan to generate code for processing chunks of tuples.

Each operator of Umbra/HyPer [99] provides two functions, `produce()` and `consume()` to generate code. On the topmost operator of an operator tree, `produce()` is called, which recursively calls the same method on its child operators. Arriving at a leaf operator, it registers pipelines for parallel execution and calls `consume()` on the parent node. Within these pipelines, the generated code processes data inside registers without overhead. Unary operators can process tuples within a pipeline, whereas binary operators have to materialise at least the result of one incoming child node first before pipelined processing begins. Operators can be classified as *pipeline friendly* or into *pipeline breakers* and *full pipeline breakers* by their behaviour of passing elements through directly, consuming all elements, or consuming and materialising all elements before producing output. An operator for gradient descent is a (full) pipeline breaker, as it accesses batches of tuples multiple times until the weights converge, whereas an operator for automatic differentiation is part of a pipeline as it just adds the partial derivatives per tuple.

5.2.2 Automatic Differentiation

Reverse mode automatic differentiation first evaluates an expression, then it propagates back the partial derivative in reverse order by applying the chain rule (see Figure 5.4). Each partial derivative is the product of the parent one (or 1 for the topmost node) and the derived function with its original arguments as input (see Figure 5.5). This allows computing each expression's derivative in one pass by reusing each subexpression (see Algorithm 2).

As Umbra compiles arithmetic expressions to machine code as well, it is perfectly suited for automatic differentiation. Similar to how an arithmetic SQL expression is compiled during code generation, we created a function that can be used to generate the expression's partial derivatives: Once a partial derivative has been compiled, its subexpressions will be cached inside an LLVM register that can be reused to generate the remaining partial derivatives. This accelerates the runtime during execution.

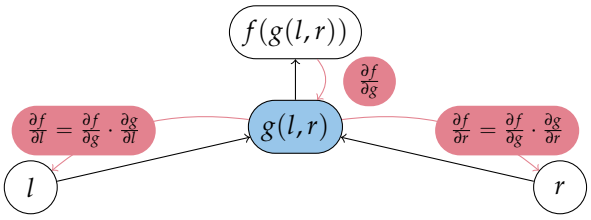


FIGURE 5.4: Reverse mode automatic differentiation: First, the function $f(g(l,r))$ gets evaluated, then each partial derivative is computed in reverse order. Each arrow represents one cached computation.

To specify the expression, we integrated lambda functions as introduced in HyPer into Umbra. Lambda functions are used to inject user-defined SQL expressions into table operators. Originally developed to parametrise distance metrics in clustering algorithms or edges for the PageRank algorithm, lambda functions are expressed inside SQL queries and allow "variation points" [102] in otherwise inflexible operators. In this way, lambda expressions broaden the application of the default algorithms without the need to modify the database

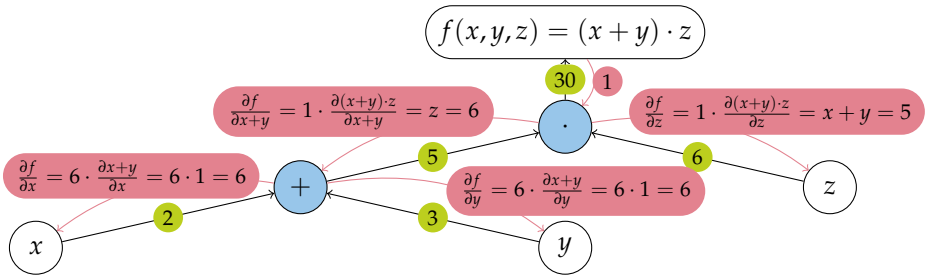


FIGURE 5.5: Automatic differentiation for $f(x,y,z) = (x+y)\cdot z$ on $x = 2, y = 3, z = 6$.

Algorithm 2 Automatic Differentiation

function DERIVE(z, z')
  if $isBinary(z)$ **then**
    if $z = x + y$ **then** DERIVE(x,z') DERIVE(y,z')
    else if $z = x - y$ **then** DERIVE(x,z') DERIVE(y,$-z'$)
    else if $z = x \cdot y$ **then** DERIVE(x,$z' \cdot y$) DERIVE(y,$z' \cdot x$)
    else if $z = X \cdot Y$ **then** DERIVE(X,$z' \cdot Y^T$) DERIVE(Y,$z'^T \cdot X$)
    else if $z = \frac{x}{y}$ **then** DERIVE(x,$\frac{z'}{y}$) DERIVE(y,$\frac{-z' \cdot x}{y^2}$)
    else if $z = x^y$ **then** DERIVE(x,$z' \cdot y \cdot x^{y-1}$) DERIVE(y,$z' \cdot x^y ln(x)$)
    else if $z = log_y(x)$ **then** DERIVE(x,$\frac{z'}{x \cdot ln(y)}$) DERIVE(y,$\frac{-z' \cdot ln(x)}{y \cdot ln^2(y)}$)
  else if $isUnary(z)$ **then**
    if $z = \sqrt{x}$ **then** DERIVE(x,$\frac{z'}{2 \cdot \sqrt{x}}$)
    else if $z = |x|$ **then** DERIVE(x,$\frac{z' \cdot x}{|x|}$)
    else if $z = sin(x)$ **then** DERIVE(x,$z' \cdot cos(x)$)
    else if $z = cos(x)$ **then** DERIVE(x,$-z' \cdot sin(x)$)
    else if $z = e^x$ **then** DERIVE(x,$z' \cdot e^x$)
  else if $isVariable(z)$ **then**
    $\frac{\partial}{\partial z} \leftarrow \frac{\partial}{\partial z} + z'$

system's core. Furthermore, SQL with lambda functions substitutes any new query language, offers the flexibility and variety of algorithms needed by data scientists, and ensures usability for non-expert database users. In addition, lambda functions allow user-friendly function specification, as the database system automatically deduces the lambda expressions' input and output data types from the previously defined table's attributes. Lambda functions consist of arguments to define names for the tuples (but whose scope is operator specific) and the expression itself. All provided operations on SQL types, even on arrays, are allowed:

$$\lambda(< \text{name1} >, < \text{name2} >, ...)(< \text{SQL expression} >).$$

```
1 select * from umbra.derivation(TABLE(select 2::float x,3::float y,6::float z),lambda(x)((x.x+x.y
      )*x.z));
2 -- x y z d_x d_y d_z
3 -- 2 3 6 6 6 5
4 select * from umbra.derivation(TABLE(select 2::float x, 3::float y, 10::float a, 10::float b),
      lambda (x)((x.a * x.x + x.b - x.y)^2));
5 -- x y a b d_x d_y d_a d_b
6 -- 2 3 10 10 540 -54 108 54
```

LISTING 5.3: Automatic differentiation within SQL.

We expose automatic differentiation as a unary table operator called `derivation` that derives an SQL expression with respect to every affected column reference and adds its value as a further column to the tuple (see Listing 5.3). We can use the operator within a recursive table to perform gradient descent (see Listing 5.4, 5.5). This eliminates the need to derive complex functions manually and accelerates the computation with a rising number of attributes, as each subexpression is evaluated only once.

```
with recursive gd (id, a, b) as (select 1,1::float,1::float UNION ALL
 select id+1, a-0.05*avg(d_a), b-0.05*avg(d_b)
 from umbra.derivation(TABLE (select id,a,b,x,y from gd,data where id<5),
 lambda (x) ((x.a * x.x + x.b - x.y)^2))  group by id,a,b)
select * from gd order by id;
```

LISTING 5.4: Gradient descent using a recursive table (automatically derived).

```
with recursive gd (id, a1, a2, a3, a4, b) as (
 select 1, 1::float, 1::float, 1::float, 1::float, 1::float
UNION ALL
 select id+1,a1-0.001*avg(d_a1),a2-0.001*avg(d_a2),a3-0.001*avg(d_a3),a4-0.001*avg(d_a4),
        b-0.001*avg(d_b)
 from umbra.derivation(TABLE (
  select * from gd, (select * from normalised tablesample reservoir (10)) where id < 51),
  lambda (x)((x.a1*x.hour + x.a2*x.day + x.a3*x.distance + x.a4*x.bearing + x.b-x.duration)^2))
 group by id, a1, a2, a3, a4, b)
select id, avg(a1*hour + a2*day + a3*distance + a4*bearing + b-duration)^2
from gd, normalised where id = 51;
```

LISTING 5.5: Using automatic differentiation for gradient descent on the taxi dataset.

5.2.3 Gradient Descent Operator

Our operator for gradient descent materialises incoming tuples, performs gradient descent and produces the optimal weights for labelling unlabelled data. The proposed operator is considered a pipeline breaker as it needs to materialise all input tuples beforehand to perform multiple training iterations. This section focuses on the operator characteristics, the design with its input queries and the actual implementation, with linear regression as an example.

Operator Design

We design an operator for gradient descent, which requires one input for the training, one for the initial weights and optionally one for the validation set, and returns the optimal weights. If no input is given as a validation set, a fraction of the training set will be used for validation. The user can set the parameters for the batch size, the number of iterations and the learning rate as arguments inside the operator call (see Listing 5.6). Figure 5.6 shows gradient descent inside of an operator tree: it expects the training dataset as parallel input pipelines and returns the optimal weights. These might serve as input for a query that labels a test dataset. In addition, SQL lambda functions, which allow users to inject arbitrary code into operators, specify the loss function to be used for gradient descent. Gradient descent benefits from generated code as it allows user-defined model functions to be derived at compile time to compute its gradient without impairing query runtime. Given two relations, $X\{[a,b]\}$ (containing the initial weights) and $Y\{[x,y]\}$ (containing the data), the loss function can be specified, for example, as the following linear combination:

$$\lambda(X,Y)(X.a \cdot Y.x + X.b - Y.y)^2.$$

```
select * from umbra.gd(TABLE (select * from data), TABLE (select 10::float a, 10::float b),
     lambda (x,y) ((y.a * x.x + y.b - x.y)^2), 1, 0.05, 10);
```

LISTING 5.6: Gradient descent as an operator: one tuple per batch, $\gamma = 0.05$, 10 iterations.

This implies three parts for the integration of gradient descent: consuming all input tuples in parallel pipelines, performing gradient descent and producing the weights in a new pipeline. This first separation is necessary, as we need to know the number of tuples in advance to determine when one training epoch ends. Specific to Umbra, we cannot assume the same number of available threads for training as for the parallel pipelines; we have to merge all materialised tuples before we start new parallel threads for the training iterations afterwards.

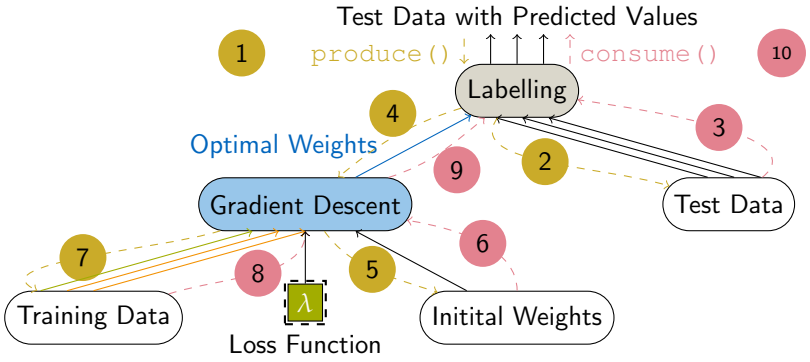


FIGURE 5.6: Operator plan inside of a database system with one operator for training and a query for predicting labels. Dashed lines illustrate code generation, solid lines compiled code. The `gradient descent` operator materialises input from parallel pipelines within local threads, performs iterations and returns the optimal weights.

Implementation

The generated code runs gradient descent iterations in parallel. Devoted to batched processing on GPUs, we deduce a parallel mini-batch gradient descent operator. First, it materialises the input tuples thread-locally (generated by `consume()`) and merges them globally. Afterwards, each thread picks one mini-batch and maintains a local copy of the global weights.

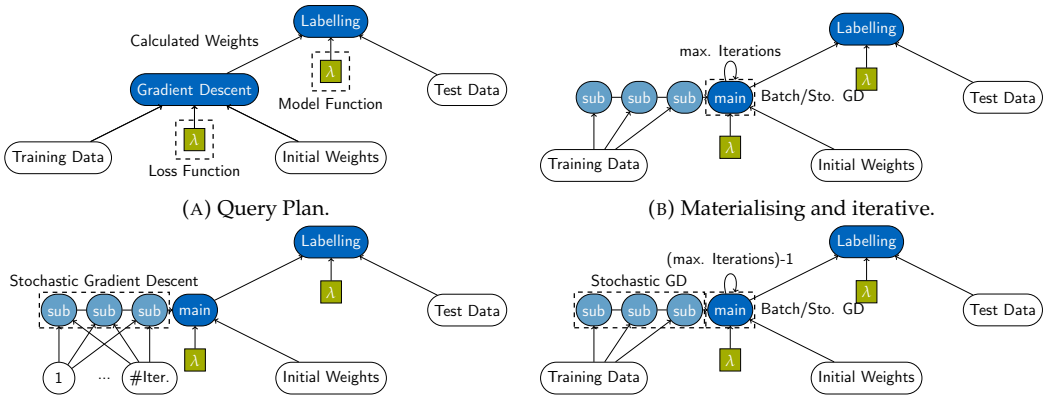
Algorithm 3 depicts the training procedure without GPU support. For simplicity, the validation phase with the corresponding validation input is omitted. Inside of the two loops (lines 5-9), one is unrolled during compile time in order to dispatch tasks to parallel threads, and one executed at runtime to manage gradient descent iterations, we can later off-load work to GPUs. Inside such a code fragment, we start as many CPU threads as GPU units are available with whom one CPU thread is associated.

Algorithm 3 Operator for mini-batch gradient descent.

1: **function** PRODUCE
2: COMPUTEGRADIENT(expression)
3: PRODUCE(inputPipeline)
4: GENERATE(mergeTuples)
5: GENERATE(while !*converged*)
6: **for** $l \in$localthreads **do**
7: GENERATE(l.updateWeights)
8: GENERATE($\vec{w} \leftarrow \sum_{l \in localthreads} \frac{l.\vec{w}}{|localthreads|}$)
9: GENERATE(whileEnd)
10: CONSUME(parent,$\vec{w}$)
11: **function** UPDATEWEIGHTS
12: GENERATE($\vec{w} \leftarrow \vec{w} - \frac{1}{|X|} \sum_{(\vec{x},y) \in X} \nabla l_{\vec{x},y}(\vec{w})$)
13: **function** CONSUME
14: GENERATE($localthread.store(\vec{x}, y)$)

Pipelining and Parallelism

We now investigate the architecture for integrating a gradient descent operator into a database system in more detail. This section does not aim for comparing the performance of stochastic and batch gradient descent but for making them fit into a database system. Therefore, we discuss different strategies regarding the methods' needs inside a database system. Batch gradient descent requires all tuples per iteration, whereas stochastic gradient descent expects one tuple at a time. The first method optimises the weights by taking the average deviation as the loss function, so all tuples need to be materialised before. When expecting



(A) Query Plan.

(B) Materialising and iterative.

(C) Pipelining and non-materialising.

(D) Combined approach.

FIGURE 5.7: Architecture for fitting gradient descent into database systems' pipelines: (a) shows the basic query plan of the gradient descent operator: a binary operator with training data, initial weights as parameters, and the lambda expression for the loss function, it returns the optimised weights, that can be passed over for labelling; (b) shows the parallelised variant, where the tuples are collected in sub-threads, unified in the main thread, where training happens in a parallel loop; (c) shows the sub-threads computing local weights without materialisation, there are as many input pipelines as iterations exist; (d) shows the combined approach, where local weights are computed initially.

only one tuple at a time, the tuples can be consumed separately or even in parallel when a synchronisation mechanism exists. Both methods work well when all tuples have been materialised before.

The intuitive way is to design gradient descent as a *full pipeline breaker* to consume all incoming tuples before producing the calculated weights. The tuples have to be materialised before, so the design allows any optimisation method to be called afterwards. For batch as well as for stochastic gradient descent, the operator's main thread performs multiple iterations until the weights converge to the loss function's minimum. Parallel loops grant distributed execution either by evaluating the gradient for each tuple independently (batch) or by updating weights as atomic global variables (stochastic). The weight synchronisation for stochastic gradient descent is based on previous work on parallelisation of optimisation problems [142, 148], which means taking the average weights after the computation is complete (and is not the focus of this section). Figure 5.7b shows the parallelism of the materialising gradient descent operator. It consists of one main thread and one sub-thread per incoming pipeline. The sub-threads consume and materialise the incoming tuples, while the main thread collects them, consumes the input weights, and minimises the loss function in parallel.

Specialised for database system's pipelines, we devise a *non-materialising* operator suited for stochastic gradient descent only. While it still is a *pipeline breaker*, it computes stochastic gradient descent in each pipeline on partitioned sets of data without materialisation. Multiple iterations are implemented by copying the underlying operator tree condoning the disadvantages of recompiling the whole subtree and the fixed number of iterations. Figure 5.7c shows the gradient descent implementation with a separate input pipeline for each iteration. For each pipeline, multiple sub-threads compute local weights. The main thread is responsible for computing the global weights after each iteration is complete.

Finally, the *combined approach* combines the benefits of parallel pipelines with the advantages of compiling the subtree only once and being able to terminate when the process has converged. First, it precomputes the weights in separate pipelines, each handling a subset of the whole data, and then performs the remaining iterations for gradient descent on the materialised tuples until convergence. Figure 5.7d shows the sub-threads computing the local weights only once for each input pipeline and materialising the tuples. The main thread computes the global weights of the first iteration, then performs the remaining iterations in parallel on the materialised tuples using either batch or stochastic gradient descent.

5.3 Neural Network

Mini-batch gradient descent can be used to train a neural network with an adjusted model function. We assume every input with m attributes serialised as one input vector $x \in \mathbb{R}^{1\times m}$ together with a categorical label $y \in L$. This corresponds to a database tuple as we later store one image as one tuple with one attribute per pixel and colour.

We consider a fully connected neural network with one hidden layer of size h, consequently we gain two weight matrices $w_{xh} \in \mathbb{R}^{m\times h}$ and $w_{ho} \in \mathbb{R}^{h\times|L|}$. With sigmoid (Equation 5.10) as activation function (applied elementwise) we obtain a model function $m_{w_{xh},w_{ho}}(x) \in \mathbb{R}^{1\times|L|}$, that produces an output vector of probabilities, also known as forward pass. The output vector is compared to the one-hot-encoded categorical label (y_{ones}). The index of the entry being one should be equal to the index of the highest probability.

$$sig(x) = (1+e^{-x})^{-1} \tag{5.10}$$

$$m_{w_{xh},w_{ho}}(x) = \underbrace{sig(\overbrace{sig(x\cdot w_{xh})}^{a_{xh}}\cdot w_{ho})}_{a_{ho}}. \tag{5.11}$$

Although neural networks can be specified in SQL-92, the corresponding query will consist of nested subqueries, that are not intuitive to create. As we have created an array data type in Chapter 3, nested subqueries can be avoided by using this data type extended by matrix algebra.

5.3.1 Neural Network in SQL-92

Expressing neural networks in SQL-92 is possible having one relation for the weights and one for the input tuples (Listing 5.7). The weights relation will contain the values in normal form as a coordinate list. If one relation contains all weight matrices, it will also contain one attribute (`id`) to identify the matrix.

We implement the forward pass in SQL as a query on the validation dataset that returns the probabilities for each category. It uses nested queries to extract the weights by an index and arithmetic expressions for the activation function. Listing 5.7 shows the forward pass with one single layer and two attributes ($m = 2$) as input. For simplicity, we use SQL functions for nested subqueries and for the sigmoid function.

```
1 create table data (a float, b float); insert into data ...
2 create table w(id int, i int, j int, val float); insert into w ...
3 -- helper functions
4 create function w_ij(id int, i int, j int) returns float language 'sql' strict as $$
5   select val from w where w.i=i and w.j=j and w.id=id $$;
6 create function sig(i float) returns float language 'sql' as $$ select 1.0/(1.0+exp(-i)); $$;
7 -- forward pass
8 select sig(i.a*w_ij(0,1,1)+i.b*w_ij(0,2,1)),sig(i.a*w_ij(0,1,2)+i.b*w_ij(0,2,2)) from data i;
```

LISTING 5.7: Forward pass for one layer within a neural network in SQL-92.

5.3.2 Neural Network with an Array Data Type

Expressing matrix operations in SQL-92 has the downside of manually specifying each elementwise multiplication. For this reason, Umbra and HyPer provide an array data type that is similar to the one in PostgreSQL and allows algebraic expressions as matrix operations.

```
1 create table wm(id int, val float[][]);
2 insert into wm select id, array_agg(name) from (select id, i, array_agg(val) as name from w
      group by id,i) j group by id;
3 -- helper function
4 create function sig (x float[]) returns float[] language 'sql' as $$
5   select array_agg(s) from (select sig(unnest) as s from unnest(x)) tmp; $$;
6 -- forward pass
7 select sig(array[[a,b]]*wm.val) from data, wm where wm.id=0;
```

LISTING 5.8: Forward pass for one layer within a neural network with an array data type.

Algorithm 4 Automatic Differentiation (Matrices)

1: **function** DERIVE(Z, Z')
2: **if** $Z = X + Y$ **then** DERIVE(X,Z') DERIVE(Y,Z')
3: **else if** $Z = X \circ Y$ **then** DERIVE(X,$Z' \circ Y$) DERIVE(Y,$Z' \circ X$)
4: **else if** $Z = X \cdot Y$ **then** DERIVE(X,$Z' \cdot Y^T$) DERIVE(Y,$Z'^T \cdot X$)
5: **else if** $Z = f(X)$ **then** DERIVE(X,$Z' \circ f'(X)$)
6: **else** $\frac{\partial}{\partial Z} \leftarrow \frac{\partial}{\partial Z} + Z'$

TABLE 5.1: Implemented activation functions and their derivatives.

Name	Abbreviation	$f(x)$	$f'(x)$
hyperbolic tangent	tanh	$\frac{e^x - e^{-x}}{e^x + e^{-x}}$	$1 - f(x)^2$
Rectified Linear Unit	relu	$\begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \geq 0 \end{cases}$	$\begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \geq 0 \end{cases}$
logistic sigmoid	sig	$\frac{1}{1+e^{-x}}$	$f(x)(1 - f(x))$

In Listing 5.8, we first construct the weight matrices from its relational representation and apply the sigmoid function on arrays as a user-defined function. Hence, the forward-pass for a single layer consists of the matrix multiplication and the sigmoid function on arrays.

An array data type allows transforming a categorical data type L into a one-hot-encoded vector even if the number of categories $|L|$ is not known at compile time. In Listing 5.9, we first create a dictionary to assign a consistent number to each category (line 3). Afterwards, we can create an array that corresponds to a one-hot-encoded vector, whose entry is one at the corresponding index with leading and subsequent zeros (line 4).

```
1 create table categories(a text); insert into categories values ('Apple'),('Egg'),('Apple');
2 with dict as (
3  select a, cast(rank() over (order by a) as int) from (select distinct a from categories) sub)
4 select categories.a, rank, array_fill(0,array[rank-1]) || 1 || array_fill(0,array[(select cast(
      count(distinct a) as int) from categories) - rank])
5 from categories, dict where categories.a=dict.a;
```

LISTING 5.9: One-hot-encoding using the SQL array data type.

5.3.3 Training a Neural Network

Automatic differentiation allows training a neural network when the derivatives of matrix multiplication [98] (see Algorithm 4) and activation functions (see Table 5.1) are implemented. To integrate the derivatives of matrix multiplication in our existing implementation, we need to extend the derivation rule for multiplications ($Z' \cdot Y^T$ instead of $z' \cdot y$ and $Z'^T \cdot X$ instead of $z' \cdot x$) and overload the transpose operator internally, so that transpose, when called on a scalar such as real numbers like floating-point values, will be ignored. Furthermore, we need the Hadamard product $X \circ Y$ (elementwise matrix multiplication) for the derivatives of the activation functions and the Hadamard power function $X^{\circ Y}$ (elementwise exponentiation) for mean squared error.

These adaptations allow our operator for automatic differentiation to train the weights for a neural network ($m = 4, h = 20, |L| = 3$) within a recursive table (see Listing 5.10): We first initialise the weight matrices with random values (line 2) and then update the weights in each iterative step (line 4) using mean squared error $(m_{w_{xh},w_{ho}}(x) - y_{ones})^{\circ 2}$ (line 7).

```
1 with recursive gd (id,w_xh,w_ho) as (
2   select 0, (select array_agg(array_agg) from generate_series(1,4),(select array_agg(random())
        from generate_series(1,20))), (select array_agg(array_agg) from generate_series(1,20),(
        select array_agg(random()) from generate_series(1,3)))
3 union all
4  select id+1, w_xh - 0.2 * avg(d_w_xh), w_ho - 0.2 * avg(d_w_ho)
5  from umbra.derivation(
6    TABLE(select * from (select * from data tablesample reservoir(50)),gd where id < 2000),
7    lambda(x)( (sig(sig(x.img**x.w_xh)*x.w_ho)-one_hot)^2 ))
8  group by id, w_ho, w_xh
9 ) select * from gd where id = 2000;
```

LISTING 5.10: Training a neural network using automatic differentiation within a recursive table.

Instead of relying on an operator for automatic differentiation, we can train a neural network by hand when applying the rules for automatic differentiation. With mean squared error, the loss is equal to the difference of labels and predicted probabilities (Equation 5.12). The factor 2 can be omitted when the learning rate γ is doubled. To train the neural network for a given input vector, we have to backpropagate the loss and update the weights as follows:

$$l_{ho} = 2 \cdot (m_{w_{xh},w_{ho}}(x) - y_{ones}), \tag{5.12}$$
$$\delta_{ho} = l_{ho} \circ sig'(a_{ho}) = l_{ho} \circ a_{ho} \circ (1 - a_{ho}), \tag{5.13}$$
$$l_{xh} = l_{ho} \cdot w_{ho}^T, \tag{5.14}$$
$$\delta_{xh} = l_{xh} \circ sig'(a_{xh}) = l_{xh} \circ a_{xh} \circ (1 - a_{xh}), \tag{5.15}$$
$$w'_{ho} = w_{ho} - \gamma \cdot a_{xh}^T \cdot \delta_{ho}, \tag{5.16}$$
$$w'_{xh} = w_{xh} - \gamma \cdot x^T \cdot \delta_{xh}. \tag{5.17}$$

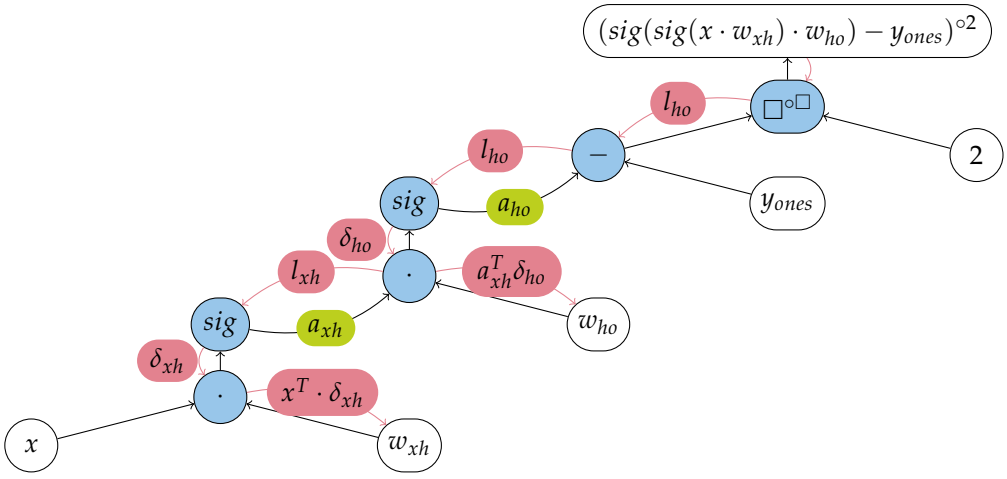


FIGURE 5.8: Automatic differentiation for $(m_{w_{xh},w_{ho}}(x) - y_{ones})^{\circ 2}$.

Figure 5.8 shows the corresponding computational graph and Listing 5.11 the corresponding code in SQL when the Hadamard product ($\circ$) is exposed as SQL operation ($**$).

```
with recursive gd (id,w_xh,w_ho) as (
  select 0, (select array_agg(array_agg) from generate_series(1,4),(select array_agg(random())
       from generate_series(1,20))), (select array_agg(array_agg) from generate_series(1,20),(
       select array_agg(random()) from generate_series(1,3)))
union all
  select id+1, w_xh - 0.2 * avg(transpose(img)*d_xh), w_ho - 0.2 * avg(transpose(a_xh)*d_ho)
  from ( select l_xh ** (a_xh ** (1- a_xh)) as d_xh, *
         from ( select d_ho*transpose(w_ho) as l_xh, *
                from (
                 select l_ho ** (a_ho ** (1- a_ho)) as d_ho, *
                 from ( select 2*(a_ho-one_hot) as l_ho, *
                        from (
                         select sig(a_xh*w_ho) as a_ho, *
                         from ( select sig(img*w_xh) as a_xh, *
                                from (select *
                                      from data tablesample reservoir(50)),gd
                                      where id < 2000))))))
  group by id, w_ho, w_xh
) select * from gd where id = 2000;
```

LISTING 5.11: Backpropagation for a neural network using a recursive table.

To process a batch of input tuples instead of a single one, the multiplications are applied on matrices instead of vectors. The multiplication during the last steps, the weight updates (Equation 5.16, 5.17), then sums up the delta for every tuple, so a simple division by the number of tuples is needed to construct the average gradient. This is helpful when vectorising mini-batch gradient descent in Section 6.1.2.

5.4 Evaluation

This section presents the evaluation of the operators in Umbra and HyPer, in detail the performance increase through automatic differentiation in Umbra, a comparison of gradient descent in HyPer to external tools and between the different architectures. All experiments were run multi-threaded on a 20-core Ubuntu 17.04 (Section 5.4.2, 5.4.3)/20.04.01 (Section 5.4.1) machine (Intel Xeon E5-2660 v2 CPU) with hyper-threading, running at 2.20 GHz with 256 GB DDR4 RAM.

5.4.1 Automatic Differentiation in Umbra

Using synthetic data, we first compare three CPU-only approaches to compute batch gradient descent (the batch size corresponds to the number of tuples) on a linear model within SQL: Recursive tables with either manually or automatically derived gradients, and a dedicated (single-threaded) operator. Figure 5.9 shows the compilation and execution time depending on the number of involved attributes. As we see, automatically deriving the partial derivatives speeds up compilation time, as fewer expressions have to be compiled, as well as execution time, as subexpressions are cached in registers for reuse. This performance benefit is also visible when the batch size, the number of iterations or the number of threads is varied (see Figure 5.10, 5.11). Furthermore, we observe the approach using recursive tables computes aggregations in parallel, which accelerates computation on many input tuples with each additional thread.

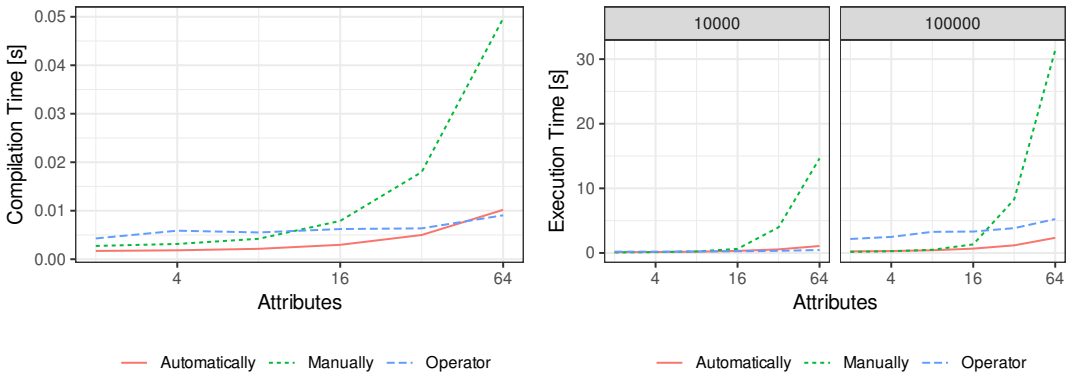


FIGURE 5.9: Linear regression: Compilation and execution time with increasing number of attributes (100 iterations, 10,000/100,000 tuples).

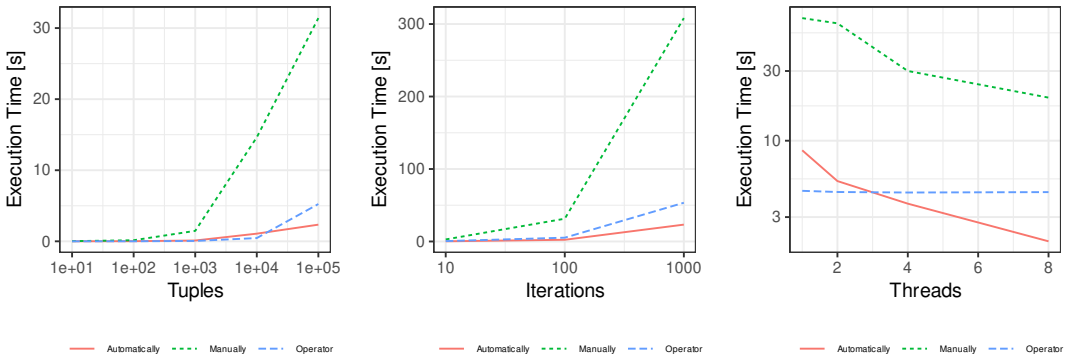


FIGURE 5.10: Linear regression: Execution time (64 attributes) with increasing number of tuples (100 iterations, 8 threads), iterations (100,000 tuples, 8 threads) or threads (100 iterations, 100,000 tuples).

The implemented derivation rules for automatic differentiation also allow deriving the gradient for arbitrary loss functions. Therefore, we also tested two different loss functions

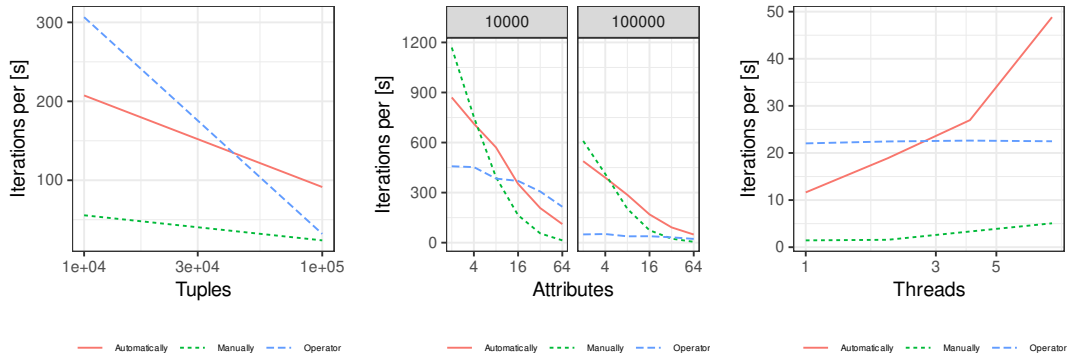


FIGURE 5.11: Iterations per second for linear regression.

for linear regression, namely mean squared error (*MSE*, Equation 5.2) and mean squared logarithmic error (*RMSLE*, Equation 5.9). Figure 5.12 shows that the choose of the loss function does not influence the runtime significantly.

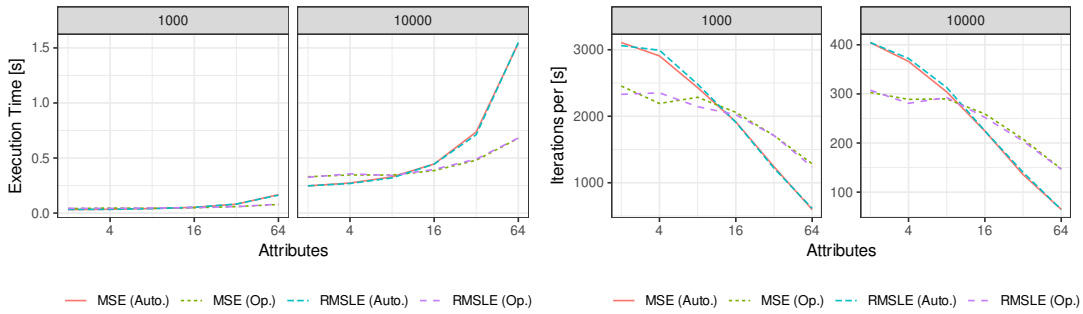


FIGURE 5.12: Different loss functions for linear regression (mean squared error *MSE*, mean squared logarithmic error *RMSLE*): with increasing number of attributes (100 iterations, 1000/10,000 tuples).

```
with recursive gd (id, a1, a2, b) as (
   select 1,1::float,1::float,1::float
UNION ALL
   select id+1,
   a1-0.5*avg(2*x1*(sig(a1*x1+a2*x2+b)-y2)*(sig(a1*x1+a2*x2+b)*(1-sig(a1*x1+a2*x2+b)))),
   a2-0.5*avg(2*x2*(sig(a1*x1+a2*x2+b)-y2)*(sig(a1*x1+a2*x2+b)*(1-sig(a1*x1+a2*x2+b)))),
   b -0.5*avg(2*    (sig(a1*x1+a2*x2+b)-y2)*(sig(a1*x1+a2*x2+b)*(1-sig(a1*x1+a2*x2+b))))
   from gd, data where id < 50 group by id,a1,a2,b)
select * from gd order by id;

with recursive gd (id, a1, a2, b) as (
   select 1,1::float,1::float,1::float
UNION ALL
   select id+1,
   a1-0.5*avg(d_a1), a2-0.5*avg(d_a2), b-0.5*avg(d_b)
   from umbra.derivation(TABLE (
      select * from gd, (select * from data) where id < 50),
      lambda (x) ((sig(b+x.a1 * x.x1 + x.a2 * x.x2 ) - x.y2)^2))
   group by id,a1,a2,b)
select * from gd order by id;

select * from umbra.gd(TABLE (select * from data), TABLE (select 1::float a1, 1::float a2, 1::
     float b), lambda (x,y) ((sig(y.b + y.a1 * x.x1 + y.a2 * x.x2) - x.y2)^2), 50, 0.5, 0);
```

LISTING 5.12: Logistic regression in SQL.

Using a recursive table, we can also solve logistic regression in SQL (see Listing 5.12). Using the sigmoid function (Equation 5.18), we know its derivative (see Table 5.1), which we can use for numeric differentiation:

$$m_{\vec{w}}(\vec{x}) = sig(\vec{x}^T \cdot \vec{w}) = \frac{1}{1 + e^{-\vec{x}^T \cdot \vec{w}}}. \tag{5.18}$$

The evaluation in Figure 5.13 shows similar results as for linear regression, although function calls for the sigmoid function slowed down the runtime especially when derived manually.

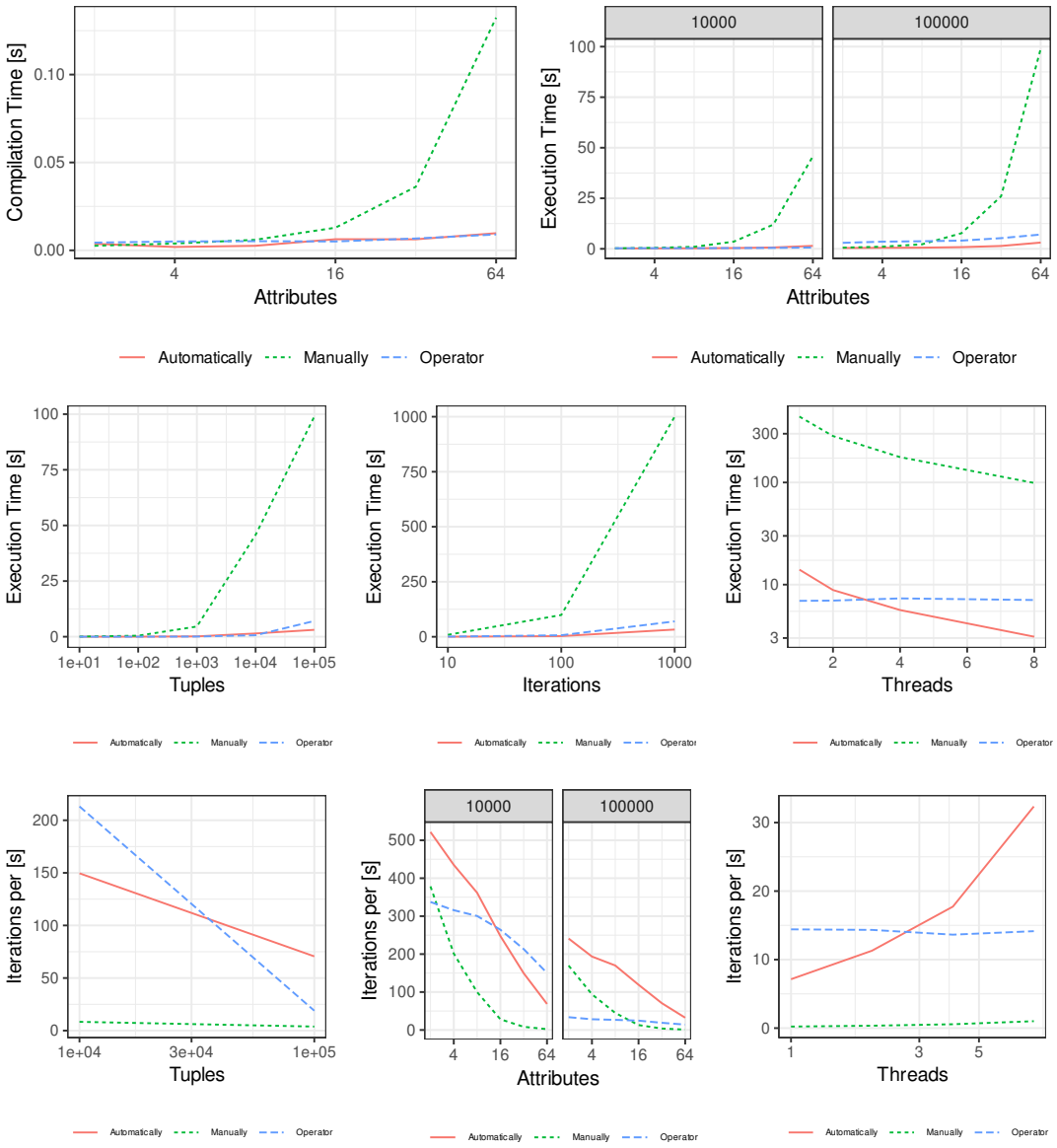


FIGURE 5.13: Logistic regression: Performance with increasing number of attributes (100 iterations, 10,000/100,000 tuples), increasing number of tuples (64 attributes, 100 iterations, 8 threads), iterations (64 attributes, 100,000 tuples, 8 threads) or threads (64 attributes, 100 iterations, 100,000 tuples).

To benchmark training a neural network, we are using the Fisher's Iris flower dataset [43] and a fully connected, two-layered neural network. In this experiment, we rather focus on the performance characteristics of the operator for automatic differentiation embedded in relational algebra than on the quality of different models. Figure 5.14 shows the compilation and execution time when training the neural network depending on the size of the hidden layer. As the size of the hidden layer depends on the weight matrices created at runtime, it does not affect the compile time. Whereas the runtime directly depends on the number of operations involved in matrix multiplications and thus increases with the size of the matrices. When we compare automatic differentiation to manually derived gradients, we applied the backpropagation rules by hand. So fewer operations were executed when using automatic differentiation, which currently computes the derivative for every variable involved. Hence, both approaches show similar performance when varying the size of the hidden layer, the

batch size, the number of iterations and threads (see Figure 5.15). Nevertheless, automatic differentiation eliminates the need for manually derived gradients and nested subqueries.

Figure 5.16 displays further experiments such as varying the number of hidden layers and the training time depending on the batch size for one epoch, so when all tuples have been processed once, using the MNIST dataset. As expected, the runtime increases with additional layers and the training time for one epoch decreases with a higher batch size. We discuss the influence of the batch size on the statistical efficiency in Section 6.3.

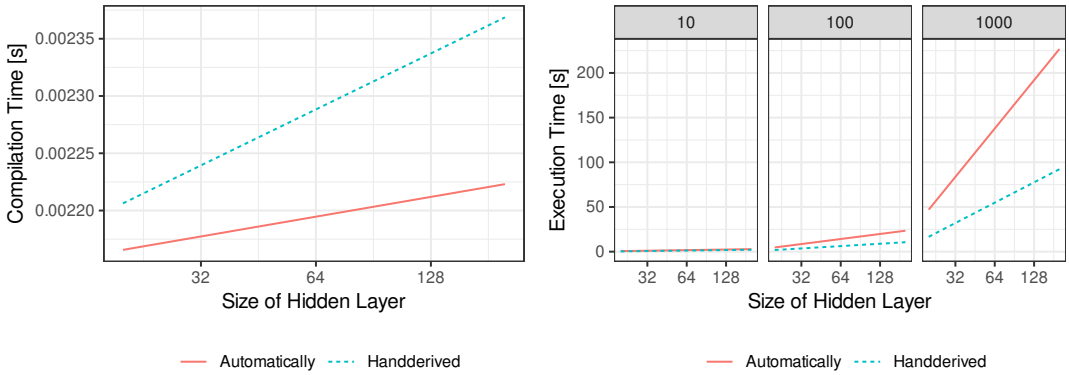


FIGURE 5.14: Neural network: Compilation and execution time with increasing size of the hidden layer (2000 iterations, batch size 10/100/1000 tuples).

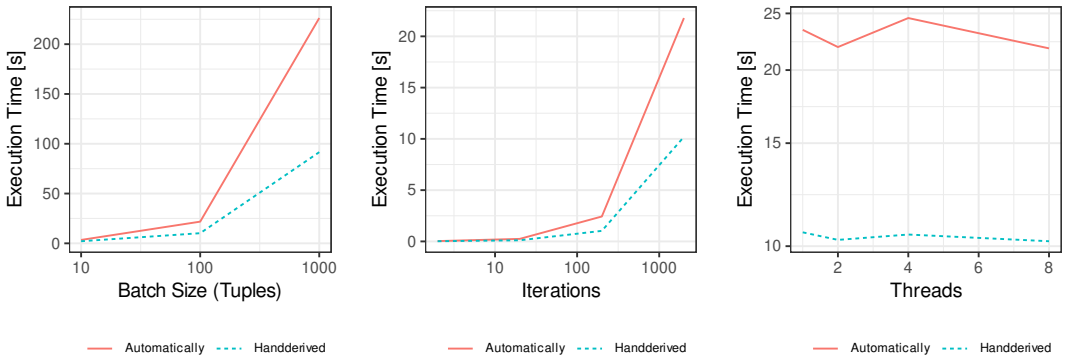


FIGURE 5.15: Neural network: Execution time (hidden layer $h = 200$) with increasing batch size (2000 iterations, 8 threads), iterations (batch size 100 tuples, 8 threads) or threads (2000 iterations, batch size 100 tuples).

5.4.2 Gradient Descent Operator in HyPer

The benchmarks for gradient descent in HyPer are based on the Chicago taxi rides dataset[1]. For each ride, this included the duration (in seconds), distance (in miles), fare, and payment type used. We used simple linear regression to predict the fare based on the trip distance, and used multiple linear regression to also consider the ride time. Finally, we used logistic regression to infer the payment type (i.e., whether or not the fare was paid by credit card) from the fare cost.

In HyPer, we ran all tests using a table operator with a call to the gradient descent library function of Simonis [128]. As a baseline, we considered two other database systems, namely MariaDB 10.1.30, PostgreSQL 9.6.8 with the MADlib v1.13 extension, as well as two dedicated tools, namely TensorFlow 1.3.0 (with GPU support enabled and disabled) and R 3.4.2. An NVIDIA GeForce GTX 1050 Ti was installed to enable TensorFlow computations on a

[1] https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew

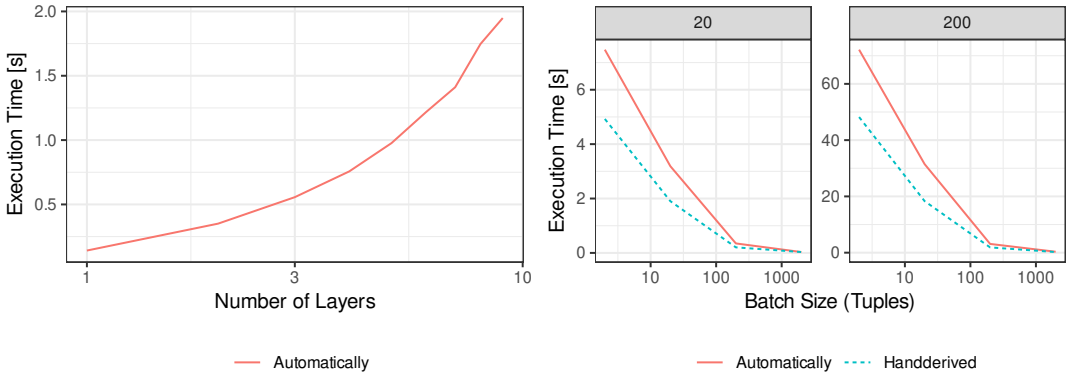
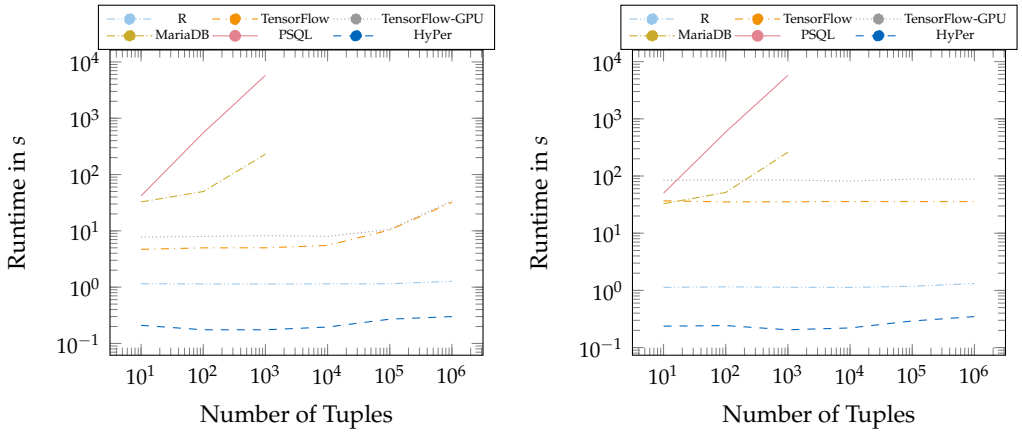


FIGURE 5.16: Neural network: Execution time depending on the number of hidden layers (each with a size of 20, batch size 2 tuples, 100 iterations) and training time for one epoch on the MNIST dataset depending on the batch size (hidden layer $h = 20/200$).

GPU. We provide the test scripts, implemented by Heyenbrock [55], for reproducibility with a data generator online[2].

All tests were carried out five times and the average runtimes recorded. A Python script managed the program calls and measured the total user time spent on each one. The runtimes include the time taken to load the data into TensorFlow and into R, and also the TensorFlow session creation time. To assume the database system as the data storage tier, the data there was considered to have already been loaded.



(A) Simple linear regression.

(B) Multiple linear regression.

FIGURE 5.17: Runtime for (a) simple and (b) multiple linear regression using gradient descent.

For the gradient descent benchmark, we evaluated linear regression, multiple linear regression, and logistic regression models, minimising the mean squared error. Each gradient descent run consisted of 5,000 iterations, with a learning rate of $7.1 \cdot 10^{-9}$.

For TensorFlow and R, we used gradient descent library functions to perform linear regression and logistic regression. For the baseline database systems, Heyenbrock [55] implemented gradient descent for both models using PL/pgSQL in PostgreSQL and procedures in MariaDB. Additionally, we benchmarked the MADlib's logistic regression function for PostgreSQL.

The results for linear regression (see Figure 5.17a, 5.17b) show that both the TensorFlow's and R's library functions run in constant time for a small number of input tuples and perform better than the hardcoded gradient descent functions in MariaDB and PostgreSQL, which scaled linearly.

[2] https://gitlab.db.in.tum.de/MaxEmanuel/regression_in_sql

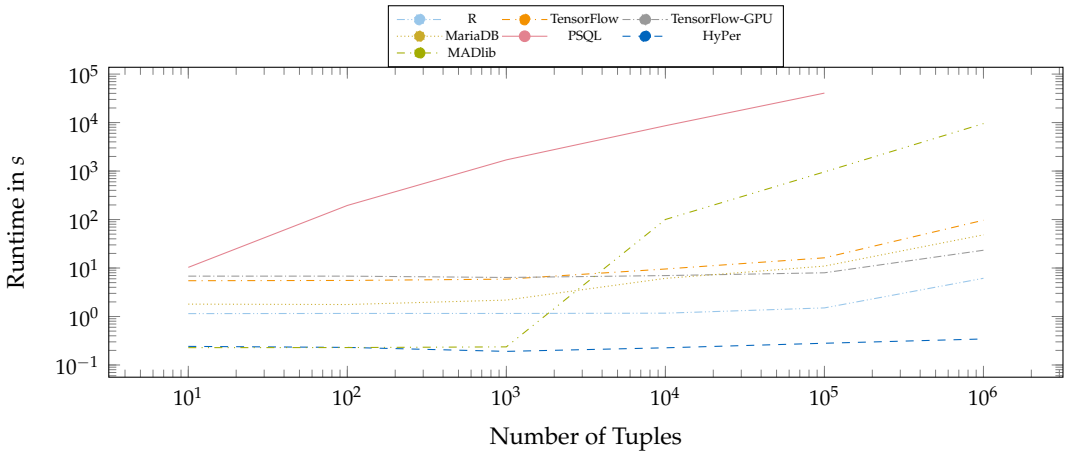
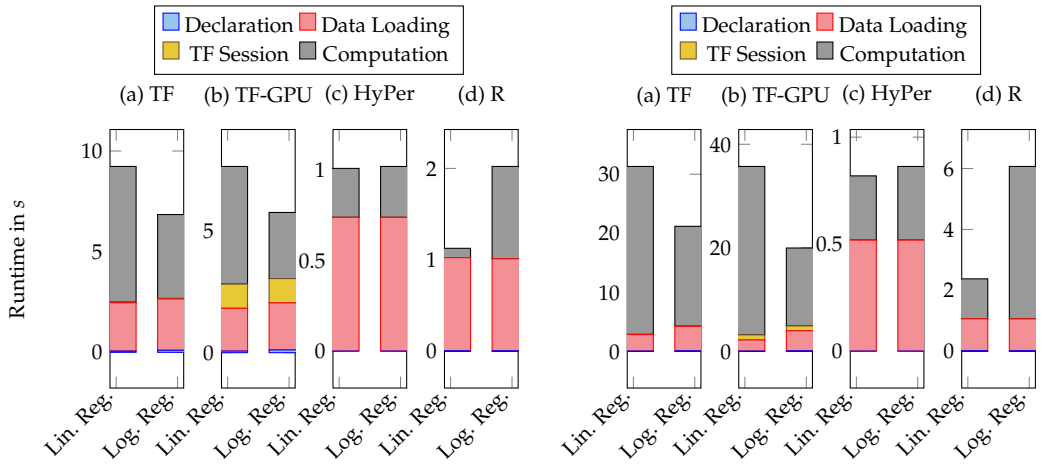


FIGURE 5.18: Runtime for logistic regression using gradient descent.

For logistic regression (see Figure 5.18), also dedicated tools' library functions, still faster than the classical database systems, scaled linearly in the input size but were slower than our approach running in HyPer.

We further investigate on the time needed for data loading and for the actual computation. In Figure 5.19, we see the runtime split up into CSV data loading, computation time, and TensorFlow's session creation. The time needed for data loading represents the time to be saved by moving computation into the database system. Also we observed that creating a session in TensorFlow takes longer when working on the GPU whereas the computation itself can be faster. This explains the better results using TensorFlow without GPU support on smaller datasets.



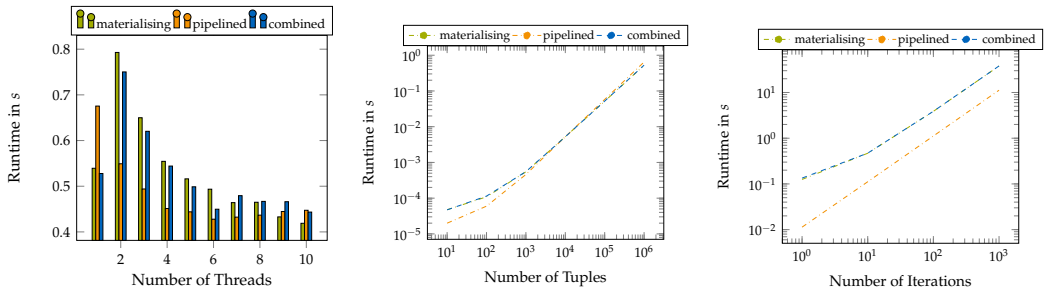
(A) 10^5 tuples. (B) 10^6 tuples.

FIGURE 5.19: Runtime split up into the different operations (declarations of variables, CSV data loading, TensorFlow's session creation, actual computation) using (a) 10^5 and (b) 10^6 tuples for simple and logistic regression by gradient descent: the time needed for data loading could be saved by doing more computations inside database systems.

In summary, much time is spent on data loading when performing computations using dedicated tools. This time can be saved without any drawbacks by shifting computation into the core of database systems. So it is worth to perform gradient descent inside database systems when suitable operators exist.

5.4.3 Pipelining within HyPer

Finally, we evaluated different approaches for integrating gradient descent into the database systems' pipelines. For this reason, we measured the performance of three different gradient descent implementations with a linear model: materialising, non-materialising (pipelined), and the combined approach. We varied the number of available threads (from one to ten, see Figure 5.20a), the training set size (from ten to 10^6 tuples, see Figure 5.20b), and iterations (from one to 100, see Figure 5.20c). The other parameters remained constant, defaulting to ten iterations, 10^6 tuples and one thread. Parallelism is enabled for the scale tests only to avoid the overhead of preparing multiple pipelines during the other tests.



(A) Scale Tests. (B) Varying the input size. (C) Varying the number of iterations.

FIGURE 5.20: Benchmarks for the different architectures of gradient descent in the database system's pipelines by varying the number of (a) available threads, (b) the input size and (c) iterations.

In all tests, the non-materialising implementation was the fastest when the methods could not terminate earlier, since all iterations are precompiled as input pipelines. The materialising and the combined implementations performed very similarly but more slowly, but had the advantage of stopping when the weights converged or reaching a predefined threshold. All three implementations behaved similarly, the runtime depends linearly on both the size of the training set and the number of iterations.

All implementations scaled linearly with the number of threads: the performance increased by about 20 % percent with each additional core. The parallelisation of the non-materialising version scaled best. When enabling parallelism but providing only one thread for execution, the non-materialising implementation performed worse than the others as the overhead of preparing sub-threads for multiple input pipelines did not pay off.

The combination of both strategies performed slightly but not significantly better in all tests. As the tuples still have to be materialised and the weights have to be computed in parallel loops, only one iteration could be saved by precomputing the weights in front. In our scenario, ten iterations, up to 10 % could be saved. But with increasing number of iterations the performance gain would be negligible. Note that one iteration per tuple corresponds to one epoch, so when all tuples of a dataset have been processed once.

To conclude, the non-materialising version appears to be the fastest when all iterations must be completed or when it is not possible to allocate enough memory for materialising the tuples. If gradient descent converges soon, a materialising approach is the most suitable, due to its ability to terminate earlier. The combination of both methods is only suited when few iterations are needed as of little performance gains otherwise. Also the non-materialising architecture can be used with stochastic gradient descent only, whereas the materialising one works with batch gradient descent as well.

5.5 Conclusion

This chapter has presented an in-database machine learning pipeline expressed in pure SQL based on sampling, continuous views and recursive tables. To facilitate gradient descent, we proposed an operator for automatic differentiation and one for gradient descent. We extended recently introduced lambda functions to act as loss functions for automatic differentiation. In comparison to handwritten derivatives, automatic differentiation as a database operator accelerated both the compile time and the execution time by the number of cached expressions. We also developed three strategies for integrating gradient descent into the pipelining concept of today's modern in-memory database systems.

A direct comparison to other relational database systems (both classical disk-based and modern in-memory systems) showed that integrating gradient descent as an operator enabled linear regression tasks to be computed more rapidly. Moreover, pure approximation tasks like logistic regression can be solved as quickly by an extended main-memory database system as by dedicated tools like TensorFlow and R, our strongest competitors.

This chapter aimed at shifting more computation into database servers. Given that modern machine learning has come to rely on tensor data and loss functions, it was important to present an architectural blueprint for combining gradient descent with tensor algebra in a modern main-memory database system. Here, we reused SQL with lambda functions as a language for specifying loss functions in a database system but, independently of the acceptance of SQL by data scientists, gradient descent will form one of the fundamental building blocks when database systems begin to take over more computational tasks, which modern hardware certainly allows.

Chapter 6

Multi-GPU Gradient Descent

Parts of this chapter have been published in [121].

In the last decade, research on database systems has focused on GPU co-processing to accelerate query engines. GPUs, initially intended for image processing and parallel computations of vectorised data, also allow general-purpose computations (GPGPU). In the context of machine learning, matrix operations [91, 37] and gradient descent [45] profit from vectorised processing available on GPUs [75]. Vectorised instructions accelerate model training and matrix computations—as the same instructions are applied elementwise, for example, in Apache Spark [53]. When a linear model is trained, vectorised instructions allow the loss as well as the gradient to be computed for multiple tuples in parallel.

This chapter explains our CUDA kernels for linear regression and neural networks, which one CPU worker thread starts once per GPU. We describe blocking and non-blocking algorithms to not hinder faster GPUs from continuing their computation while waiting for the slower ones to finish. To synchronise multiple workers, we either average the gradient after each iteration or maintain local models as proposed by Crossbow [64]. We adapt this synchronisation technique to maximise the throughput of a single GPU as well. As a novelty, we implement learners at hardware level—each associated to one CUDA block—to maximise the throughput on a single GPU. This chapter's contributions are

- fine-tuned GPU kernels that maximise the GPU specific throughput even for small and medium batch sizes,
- a performant scheduler for accelerating gradient descent iterations on multiple GPUs in parallel,
- and an evaluation of strategies for synchronising gradient descent on processing units with different throughput.

Based on automatic differentiation and on a dedicated operator for gradient descent (Chapter 5), we want to generate code directly for GPUs. The code processes mini batches on GPUs and synchronises multiple learners on a single GPU (Section 6.1) as well as parallel workers on multiple devices (Section 6.2). We will evaluate CPU- and GPU-only approaches in terms of performance and accuracy using a NUMA-server cluster with multiple GPUs (Section 6.3).

6.1 Kernel Implementations

Developing code for NVIDIA GPUs requires another programming paradigm, as computation is vectorised to parallel threads that perform the same instructions simultaneously. Each GPU device owns one global memory (device memory) and an L2 cache. Core components are streaming multiprocessors with an attached shared memory (see Figure 6.1) to execute specialised programs for CUDA devices (*kernels*). In these kernels, every thread receives a unique identifier, which is used to determine the data to process. 32 threads in a bundle are called a *warp*, multiple warps form a *block* and threads inside a block can communicate through shared memory and interfere with each other. To interfere with other threads, shuffling can be used to share or broadcast values with one or more threads within a warp.

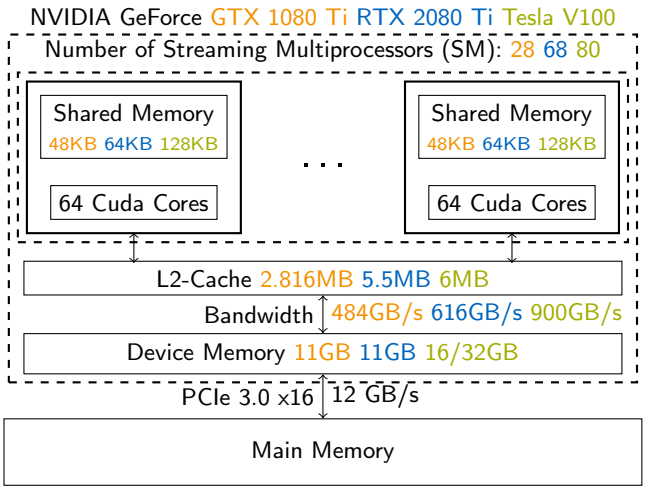


FIGURE 6.1: Simplified GPU architecture for NVIDIA GeForce GTX 1080 Ti (orange), RTX 2080 Ti (blue) and Tesla V100 (green): Each GPU transfers data via PCIe x16 from main-memory to its global/device memory. Multiple CUDA cores sharing the L1 cache (shared memory) are grouped to one streaming multiprocessor, its number is GPU specific. In-between lies the L2 cache.

To off-load gradient descent iterations to NVIDIA GPUs, we generate specialised kernels. In detail, we have to map batches to blocks; we can vary the number of threads per block and explicitly cache values in shared memory. In the following sections, we describe our kernel implementations for gradient descent with linear regression and a fully-connected neural network, and we will introduce independent learners at block-size granularity.

6.1.1 Linear Regression

As linear regression is not a compute-bound but a memory-intensive application, we initially transfer as much training data into device memory as possible. If data exceeds the memory and more GPUs are available for training, we will partition the data proportionally to multiple devices. Otherwise, we reload the mini-batches on demand.

Each thread handles one input tuple and stores the resulting gradient after each iteration in shared memory. Each iteration utilises all available GPU threads, wherefore the size of a mini-batch must be greater or equal to the number of threads per block, to ensure that compute resources are not wasted. When the batch size is larger than a block, each thread processes multiple tuples and maintains a thread-local intermediate result, which does not require any synchronisation with other threads. After a mini-batch is processed, shuffling operations summarise the gradient to compute the average for updating the weights (tree reduction).

6.1.2 Neural Network

Our initial approach was to adapt the gradient descent kernel for linear regression to train a neural network and to spread each tuple of a batch to one thread. As training neural networks is based on matrix operations, we rely on libraries for basic linear algebra subprograms for CUDA devices (cuBLAS[1]), which provide highly optimised implementations. Our implementation uses the cuBLAS API for all operations on matrices or vectors. For example, the forward pass in a neural network uses matrix-vector multiplications (`cublasDger()`) for a single input tuple and, when a mini-batch is processed, matrix-matrix multiplications respectively (`cublasDgemm()`). To apply and derive the activation function, handwritten kernels are used that vectorise over the number of attributes. These kernels plus the library calls plus handwritten code build the foundation for parallelising to multiple GPUs.

[1] https://docs.nvidia.com/cuda/cublas

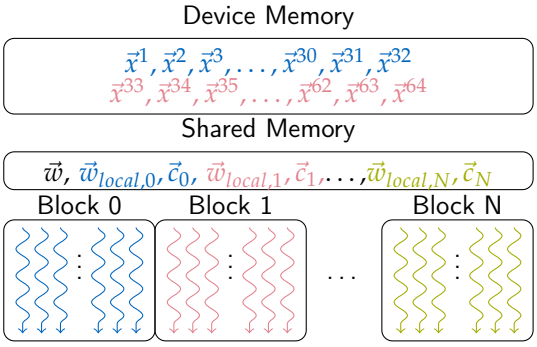


FIGURE 6.2: Multiple learners per GPU: Each block corresponds to one learner, each learner maintains local weights $\vec{w}_{local}$ and the difference $\vec{c}_{local}$ to the global weights $\vec{w}$. Each input tuple is stored in device memory and is scheduled to one GPU thread.

6.1.3 Multiple Learners per GPU

To utilise all GPU threads even with small batch sizes, we implement multiple workers on a single GPU. These are called *learners* [64] and ensure a higher throughput. Crossbow offers a coarse-grained approach as every learner launches multiple kernels, which limits its overall number. By contrast, our light-weight approach launches only one instance of a fine-grained kernel for one entire GPU. This enables full utilisation of the GPU as the number of learners could be much higher.

In our implementation (see Figure 6.2), each learner corresponds to one GPU block. We can set the block size adaptively, by which the number of learners results. Consequently, one learner works on batch sizes of at least one warp, that is the minimum block size with 32 threads, or multiple warps. Hence, the most learners that are allowed is the number of warps that can be processed per GPU.

After each learner has finished its assigned batches, the first block synchronises with the other ones to update the global weights. But for multi GPU processing as well as for multiple learners per GPU, we need to synchronise each unit.

6.2 Synchronisation Methods

As we intend to run gradient descent in parallel on heterogeneous hardware, we have to synchronise parallel gradient descent iterations. Based on a single-threaded naive gradient descent implementation, we propose novel synchronisation methods to compare their performance to existing ones and benchmark different hardware.

The naive implementation uses a constant fraction of its input data for validation and the rest for training. The training dataset is split into fixed-sized mini-batches. After a specified number of mini-batches but no later than after one epoch when the whole training set has been processed once, the loss function is evaluated on the validation set and the current weights. The minimal loss is updated and the next epoch starts. We terminate when the loss falls below a threshold l_{stop} or a maximum number of processed batches ctr_{max}. Also, we terminate if the loss has not changed within the last 10 iterations.

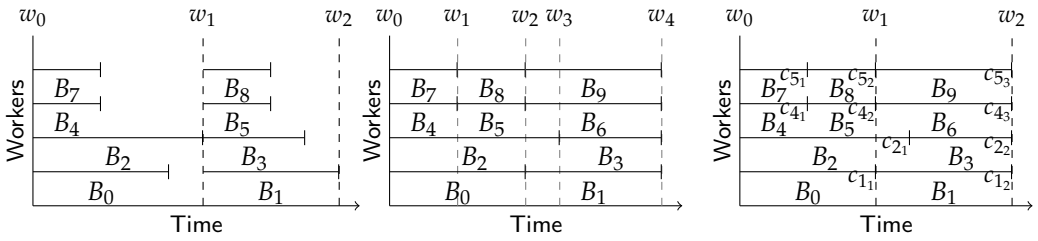
Based on the naive implementation, this section presents three parallelisation techniques, a blocking but synchronised one and two using worker threads with multiple local models or only one global one.

6.2.1 Synchronised Iterations

At the beginning of each synchronised iteration, we propagate the same weights with an individual mini-batch to the processing unit. After running gradient descent, the main worker collects the calculated gradients and takes their average to update the weights.

Algorithm 5 shows this gradient descent function, taking as input the labelled dataset X, a learning rate γ, the batch size n and the parameter ctr_{max} that limits the number of iterations. In each iteration, multiple parallel workers pick a mini-batch and return the locally

computed gradient. Afterwards, the weights are updated. For simplicity, the validation pass is not displayed: When the calculated loss has improved, the minimal weights together with the minimal loss are set and terminate the computation when a minimal loss l_{min} has been reached.



(A) Synchronised threads (B) Worker threads (global updates) (C) Worker threads (local models)

FIGURE 6.3: Scheduling mini-batches on four different workers: (a) shows worker threads whose weights are synchronised globally after each iteration and whose averaged gradient is used to update the global weights w; workers are idle when others have still not finished. (b) shows worker threads that update weights globally without any synchronisation; each worker is responsible for fetching the next batch on its own. To overcome race conditions, the worker threads in (c) maintain their local model that is synchronised lazily when every worker is done with at least one iteration.

Algorithm 5 Synchronised.

```
1:  function GD(X,γ, ctr_max, n)
2:      w⃗ ← (0, ..., 0)
3:      ctr ← 0
4:      while ctr < ctr_max do
5:          for t ∈ [#workers] do                          } in parallel
6:              batch ← ATOMIC_FETCH_ADD(ctr,1)            }
7:              X′ ← GETBATCH(batch,X,n)                   }
8:              g⃗_t ← γ∇l_{X′}(w⃗)                         }
9:          for t ∈ [#workers] do
10:             w⃗ ← w⃗ − γg⃗_t
```

When synchronising a global model after each iteration, workers who may have finished their mini-batches earlier, are idle and waiting for input (see Figure 6.3a). To maximise the throughput, independent workers have to fetch their mini-batches on their own. These independent workers either require local weights to be synchronised frequently (see Figure 6.3c) or update global weights centrally (see Figure 6.3b).

6.2.2 Worker Threads with Global Updates (Bulk Synchronous Parallel)

In Algorithm 6, we see worker threads that fetch the next batch independently and update a global model. Each worker increments a global atomic counter as a batch identifier and selects the corresponding batch consisting of the attributes and the labels. The current weights are used to compute the gradient; afterwards, the weights are updated globally. Besides, the first thread is responsible for managing the minimal weights. Assuming a low learning rate, we suppose the weights are changing marginally and we omit locks similar to HogWild [105]. Otherwise, the critical section (line 5)—gradient calculation and weights update—has to be locked, which would result in a single-threaded process as in Algorithm 5.

6.2.3 Worker Threads with Local Models (Model Average)

To overcome race conditions when updating the weights, we adapt local models known from Crossbow [64] to work with worker threads. Crossbow adjusts the number of parallel so-called learners adaptively to fully utilise the throughput on different GPUs. Each learner

Algorithm 6 Worker threads (global updates).

```
1:  function RUN(X, w⃗, ctr, ctr_max, γ)
2:      while ctr < ctr_max do
3:          batch ← ATOMIC_FETCH_ADD(ctr,1)
4:          X' ← GETBATCH(batch,X,n)
5:          w⃗ ← w⃗ − γ∇l_X'(w⃗)            } critical section
6:  function GD(X,γ, ctr_max, n)
7:      w⃗ ← (0, ..., 0)
8:      ctr ← 0
9:      for t ∈ [#workers] do
10:         RUN(X, w⃗, ctr, ctr_max, γ)     } in parallel
```

maintains its local weights and the difference from the global weights. A global vector variable for every learner t called corrections $\vec{c}_t$ stores this difference, divided by the number of all learners. In each iteration, the weights are updated locally and these corrections are subtracted. After each iteration, the corrections of all learners are summed up to form the global weights.

Algorithm 7 shows its adaption for worker threads. The main thread manages the update of the global model (line 11) that is the summation of all corrections. The critical section now consists of the computation of the corrections (lines 7-9) only, so the gradient can be computed on multiple units in parallel.

Algorithm 7 Worker threads (local models).

```
1:  function RUN(X, w⃗, ctr, ctr_max, γ)
2:      w⃗_local ← w⃗
3:      while ctr < ctr_max do
4:          batch ← ATOMIC_FETCH_ADD(ctr,1)
5:          X' ← GETBATCH(batch,X,n)
6:          g⃗ ← γ∇l_X'(w⃗_local)
7:          c⃗_threadid ← (w⃗_local − w⃗)/t
8:          w⃗_local ← w⃗_local − γg⃗ − c⃗_threadid
9:          c⃗_threadid ← (w⃗_local − w⃗)/t
10:         if threadid = 0 then
11:             w⃗ ← w⃗ + Σ_{t∈[#workers]} c⃗_t
                                            } critical section
12: function GD(X,γ, ctr_max, n)
13:     w⃗ ← (0, ..., 0)
14:     ctr ← 0
15:     for t ∈ [#workers] do
16:         RUN(X, w⃗, ctr, ctr_max, γ)     } in parallel
```

6.3 Evaluation

We tested on servers with four Intel Xeon Gold 5120 processors, each with 14 CPUs (2.20 GHz) running Ubuntu 20.04.01 LTS with 256 GiB RAM. Each server is connected to either four GPUs (NVIDIA GeForce GTX 1080 Ti/RTX 2080 Ti) or one NVIDIA Tesla V100-PCIE-32GB. We benchmark linear regression with synthetic data and the New York taxi dataset, and a feed-forward neural network with a single hidden layer for image recognition (see Table 6.1). We take 2.65 GiB of the New York taxi dataset[2] (January 2015), on which we perform linear regression to forecast the taxi trip duration from the trip's distance and bearing, the day and the hour of the ride's beginning (four attributes).

6.3.1 Linear Regression

We measure the performance and the quality of the different parallelisation models on the CPU as well as the GPU according to the following metrics:

1. *Throughput* measures the size of processed tuples per time. It includes tuples used for training as well as for validation.
2. *Time-to-loss*, similarly to *time-to-accuracy* [28] for classification problems, measures the minimal loss on the validation dataset depending on the computation time.
3. *Tuples-to-loss* describes how many tuples are needed to reach a certain minimal loss. In comparison to *time-to-loss*, it is agnostic to the hardware throughput and measures the quality of parallelisation and synchronisation methods.

TABLE 6.1: Datasets used with linear regression and a neural network respectively.

	#attr.	*#training*	*#validation*
New York Taxi	4 + 1	61,664,460	15,416,115
Synthetic	99 + 1	10	10
MNIST	784 + 1	60,000	10,000
Fashion-MNIST	784 + 1	60,000	10,000

We perform gradient descent with a constant learning rate of 0.5 to gain the optimal weights. After a predefined validation frequency, every 3,000 batches, the current loss is computed on a constant validation set of 20 % the size of the original one. We vary the hyper-parameters of our implementation, i.e., the batch size and the number of workers. A thread records the current state every second to gather loss metrics.

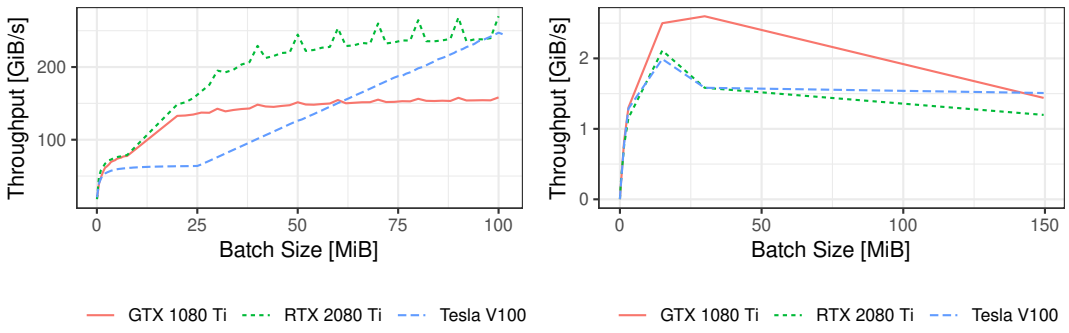
Throughput vs. Statistical Efficiency

To measure the throughput for linear regression on different hardware, we consider batch sizes of up to 100 MiB. We compare the performance of our kernels to that when stochastic gradient descent of the TensorFlow (version 1.15.0) library is called (see Figure 6.4).

The higher the batch size, the better the throughput when running gradient descent on GPUs as all concurrent threads can be utilised. Hardware-dependent, the maximal throughput converges to either 150 GiB/s (GeForce GTX 1080 Ti), 250 GiB/s (GeForce RTX 2080 Ti) or more than 300 GiB/s (Tesla V100). As developed for batched processing, our dedicated kernels (see Figure 6.4a) can exploit available hardware more effectively than the Python implementation (see Figure 6.4b). As the latter calls stochastic gradient descent, this excels on better hardware only when a larger model has to be trained.

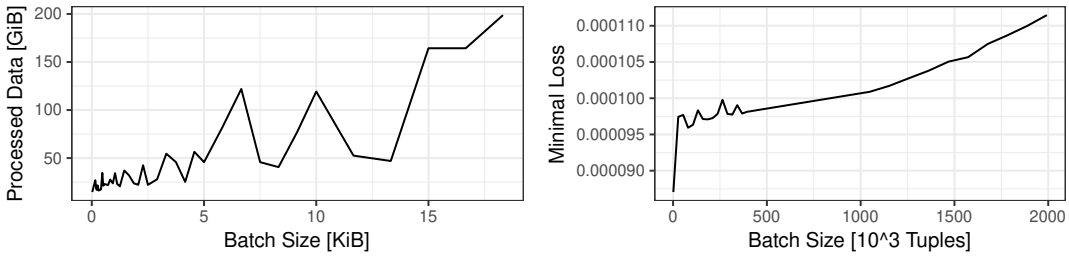
Nevertheless, training with large batch sizes does not imply statistical efficiency in terms of the volume of processed data that is needed for convergence (see Figure 6.5a) and the lowest reachable minimal loss (see Figure 6.5b). For that reason, to allow the highest throughput even for small batch sizes, we implement multiple learners per GPU.

[2] https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2015-01.csv



(A) Our implementation. (B) Using TensorFlow.

FIGURE 6.4: Throughput of (a) the dedicated kernels and (b) using the TensorFlow library.

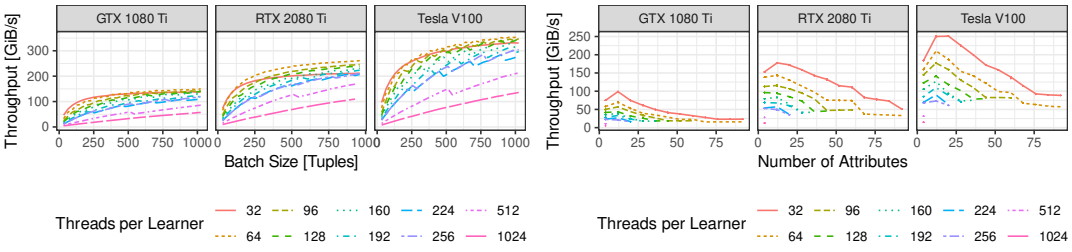


(A) Data to converge. (B) Minimal loss.

FIGURE 6.5: Statistical efficiency for linear regression: (a) volume of processed data needed to converge and (b) minimal reachable loss depending on the batch size.

Multiple Learners per GPU

As the GPU is only fully utilised when the number of concurrently processed tuples is greater or equal to the number of parallel GPU threads, we benchmark multiple learners per GPU. As each learner corresponds to one GPU block consisting of a multiple of 32 threads, our implementation allows the highest throughput for every batch size, as a multiple of the block size. Therefore, we vary the number of threads per block (equal to a learner) between 32 and 1,024 and measure the throughput dependent on the batch size in multiples of 32 threads.



(A) Varying the batch size (four attributes). (B) Number of attr. (64 tuples per batch), synthetic data.

FIGURE 6.6: Throughput with multiple learners per GPU: A smaller number of threads per learner allows the maximum throughput even for small batch sizes when small batches are processed in parallel.

The observation in Figure 6.6 corresponds to the expectation that a small number of threads per learner allows a higher throughput for small batch sizes. When the batch size is equal to a multiple of the chosen number of threads, the throughput reaches a local maximum. Otherwise, the GPU is underutilised. These local maxima are visible as spikes in all

curves except for 32 threads per block, as we increase the batch size by 32 tuples. Nevertheless, on all devices, the throughput soon converges at the possible maximum, which shows the efficiency of learners in the granularity of GPU warps.

Scalability

When running gradient descent in parallel, we benchmark the four implementations for synchronising weights: no synchronisation with global updates (*global updates*), maintaining local models either with locking of the critical section (*local models (locks)*) or without locking (*local models (dirty)*), or synchronised updates that block until every worker has finished (*synchronised (blocking)*). We ran the experiments on the CPU as well as the GPU.

When parallelising on the CPU, each additional thread allows a linear speed-up when no synchronisation takes place (see Figure 6.7a). Maintaining local models costs additional computation time, which results in a lower throughput. Obviously, locks slow down the speed up, and blocking threads cause underutilisation.

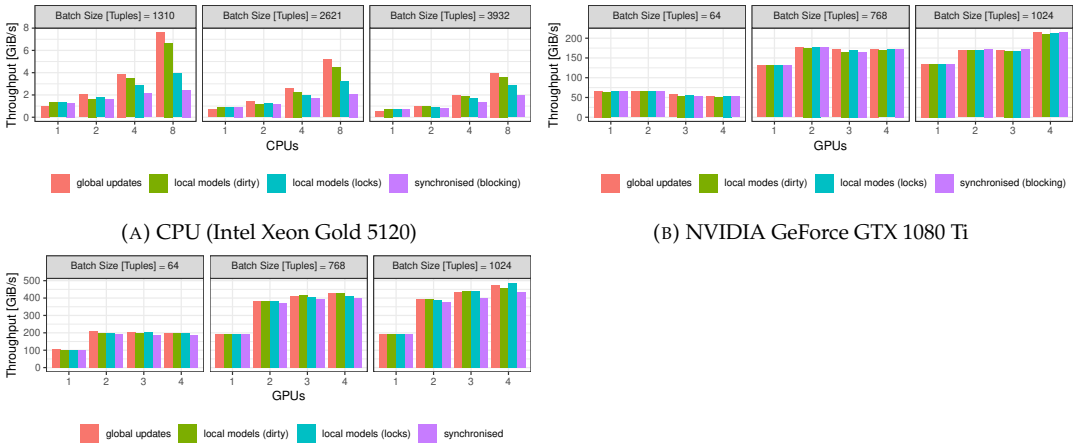


FIGURE 6.7: Scale-up for linear regression on (a) multiple CPUs, (b) multiple NVIDIA GeForce GTX 1080 Ti or (c) multiple NVIDIA GeForce GTX 2080 Ti.

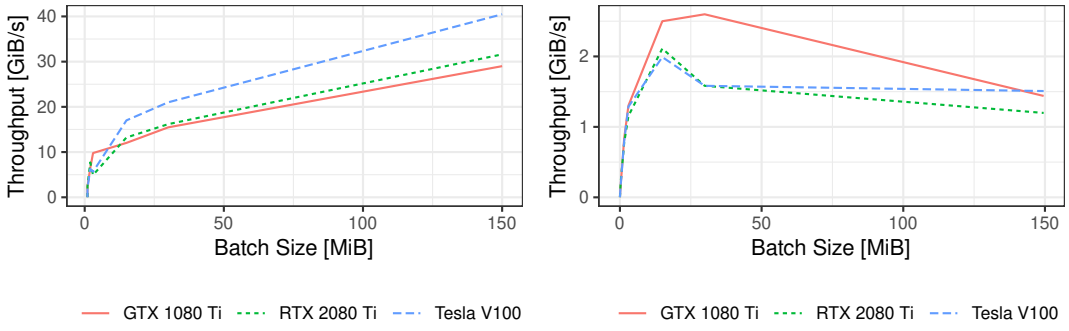
Whereas parallelising for GPUs behaves differently (see Figure 6.7b/c): the larger the batch size, the higher the scale-up. This is obvious, as less synchronisation is necessary for larger batch sizes and the parallel workers can compute the gradients independently. Also on GPUs, the implementation without any synchronisation and global updates scales best, even though not as linearly as on CPUs. In all implementations, one additional GPU allows a noticeably higher throughput. Maintaining local models requires inter-GPU communication of the local corrections to form the global weights, which decreases the performance significantly with the third additional device. To minimise this effect, the weight computation could be split up hierarchically.

6.3.2 Neural Network

To benchmark the feed-forward neural network, we perform image classification using the MNIST and Fashion-MNIST [141] dataset. We train the neural network with one hidden layer of size 200 to recognise a written digit (Fashion-MNIST: a piece of clothing) given as a single tuple representing an image with 784 pixels. We take 0.025 as learning rate, perform a validation pass every epoch and measure the throughput and the time to reach a certain accuracy (with the loss defined as the number of incorrectly classified tuples).

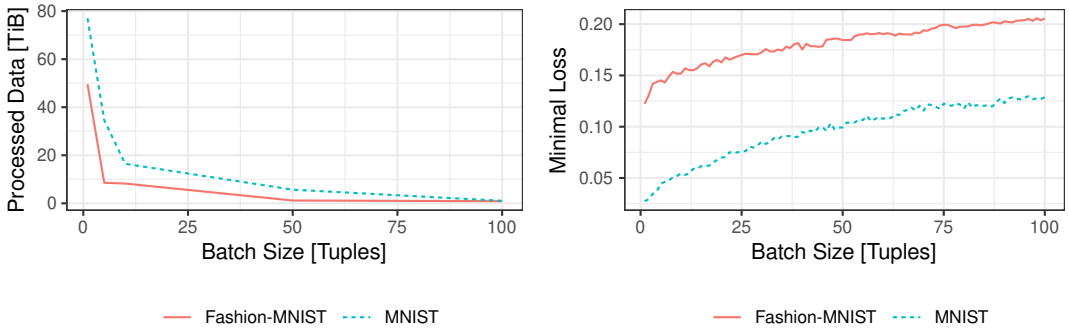
Throughput vs. Statistical Efficiency

Even though stochastic gradient descent using Keras (version 2.2.4) with TensorFlow allows a higher bandwidth than for linear regression due to more attributes per tuple (see Figure 6.8b), our implementations, which call the cuBLAS library, process tuples batch-wise, which results in a higher bandwidth. As training a neural network is compute-bound involving multiple matrix multiplications, the throughput is significantly lower than for linear regression (see Figure 6.8a), but allows a higher throughput, the larger the batch size.



(A) Own implementation. (B) Using Keras.

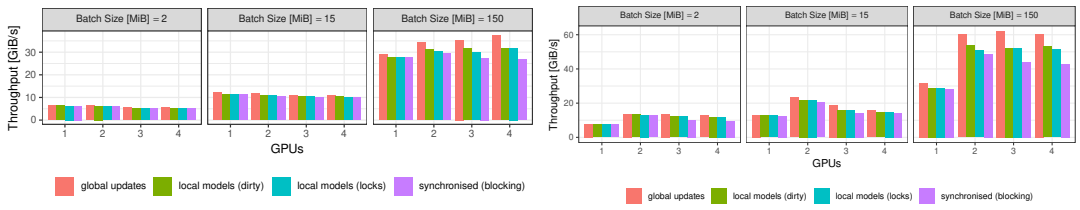
FIGURE 6.8: Throughput of the implementation (a) using cuBLAS and (b) using Keras when training a neural network with the MNIST dataset.



(A) Data to converge. (B) Minimal loss.

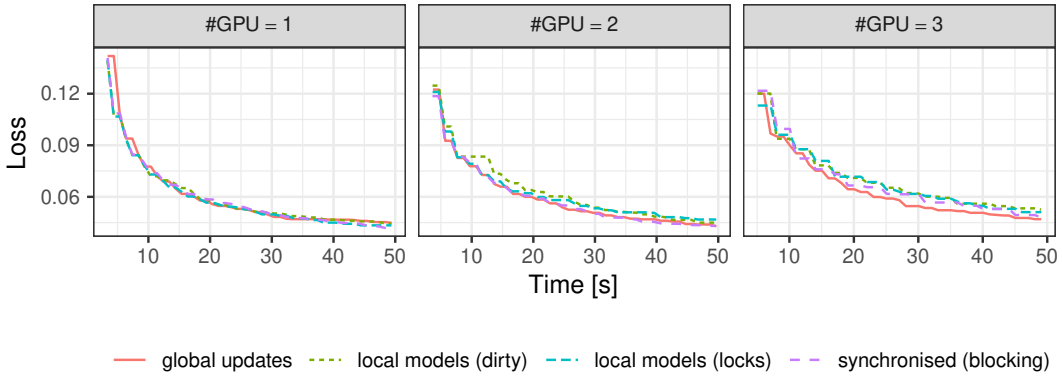
FIGURE 6.9: Statistical efficiency for the neural network: (a) volume of processed data needed to converge and (b) minimal reachable loss depending on the batch size.

As is the case for linear regression, training models with small batch sizes results in a higher accuracy (see Figure 6.9b). This once again makes the case for multiple learners per single GPU. Nevertheless, the larger the chosen batch size is, the faster training iterations converge (see Figure 6.9a).

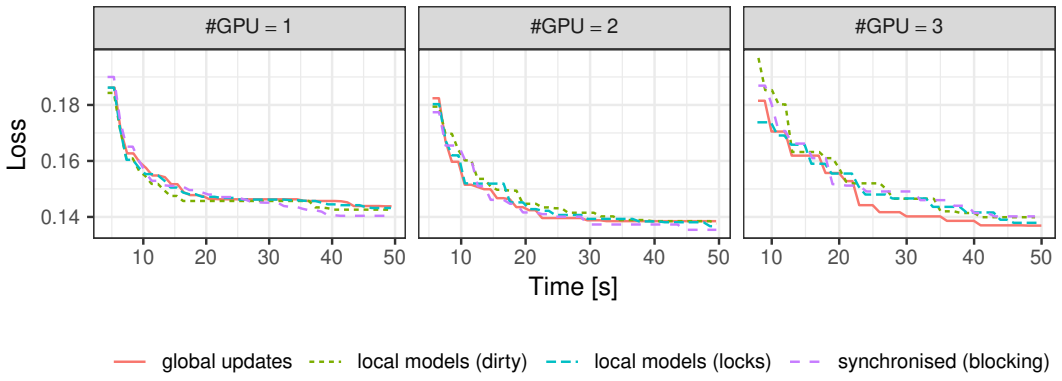


(A) NVIDIA GeForce GTX 1080 Ti (B) NVIDIA GeForce RTX 2080 Ti

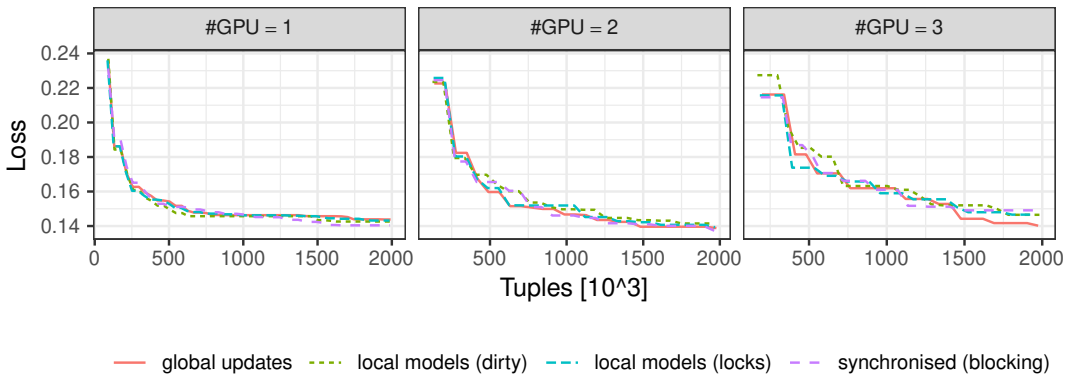
FIGURE 6.10: Scale-up for training a neural network (with the MNIST dataset).



(A) Time-to-loss: MNIST dataset



(B) Time-to-loss: Fashion-MNIST dataset



(C) Tuples-to-loss: Fashion-MNIST dataset

FIGURE 6.11: Time-to-loss when training the neural network for the (a) MNIST and (b) Fashion-MNIST dataset with a batch size of 5 tuples (NVIDIA GeForce GTX 2080 Ti). For Fashion-MNIST, also tuples-to-time (c) is provided.

Scalability

The scalability of parallel workers computing backpropagation resembles the scalability for training linear regression on GPUs: one additional worker increases the throughput, for any further workers, the inter-GPU communication decreases the runtime (see Figure 6.10). For small batch sizes, training on two GPU devices has the best results, while for larger batch sizes, every additional device allows a higher throughput.

Time/Tuples-to-loss

Regarding the time to reach a certain accuracy (see Figure 6.11), all implementations perform similarly when running on a single worker. As the MNIST dataset converges fast, adding a GPU device for computation has no significant impact. Whereas the Fashion-MNIST dataset converges slower, the higher throughput when training with an additional worker results in the minimal loss being reached faster. We train with a small batch size as it allows faster convergence. Hereby, a scale-up is only measurable when training with up to two devices.

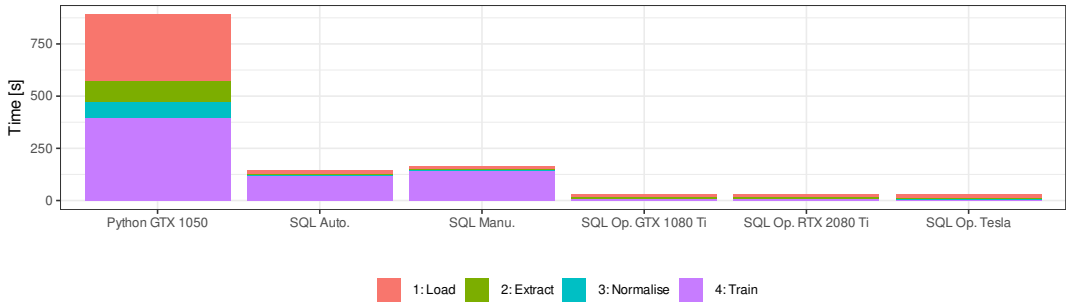
6.3.3 End-to-End Analysis

Figure 6.12 compares the time needed to train one epoch (New York taxi data: $13 \cdot 10^6$ tuples, MNIST: $6 \cdot 10^4$ tuples) within a complete machine learning pipeline in Python using Keras to a corresponding operator tree within the database system Umbra, either using a recursive table with a manually or automatically derived gradient or using an operator that off-loads to GPU. The pipeline consists of data loading from CSV, feature extraction (only for the New York taxi data) and normalisation either with NumPy or SQL-92 queries, and training. Note that the measurements using the MNIST dataset were performed on a Ubuntu 20.04 LTS machine with four cores of Intel i7-7700HQ CPU, running at 2.80 GHz clock frequency each, 16 GiB RAM and equipped with an NVIDIA GeForce GTX 1050.

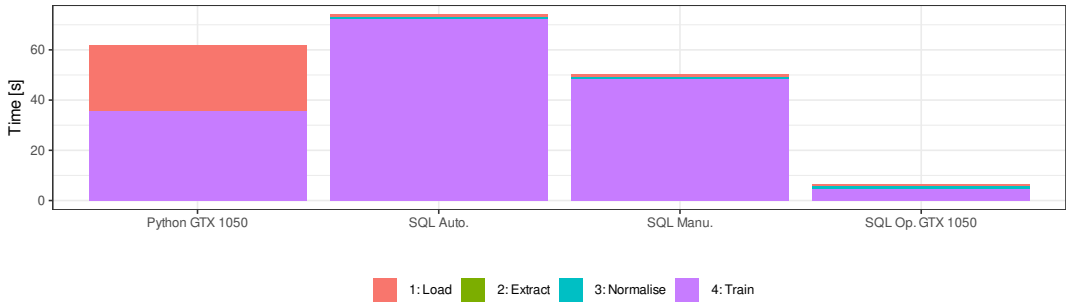
We observe that much time is spent on data loading and preprocessing. These tasks are either no longer required if the data is already stored inside the database system, or can easily be processed in parallel pipelines. Furthermore, gradient descent using recursive tables showed comparable performance to library functions used, which is still outperformed by our operator that off-loads training to GPU.

6.4 Conclusion

This chapter proposed accelerating in-database gradient descent by off-loading training to GPU units. Therefore, we have implemented training algorithms as GPU kernels and fine-tuned learners at hardware level to increase the learning throughput. These kernels were integrated inside the code-generating beyond main-memory database system Umbra. Our evaluation benchmarked GPU kernels on different hardware, as well as parallelisation techniques with multiple GPUs. The evaluation has shown that GPUs traditionally excel the bigger the chosen batch sizes, which was only worthwhile when a slow-converging model was being trained. In addition, larger batch sizes interfered with statistical efficiency. For that reason, our fine-tuned learners at hardware level allowed the highest possible throughput for small batch sizes equal to a multiple of a GPU warp, so at least 32 threads. Our synchronisation techniques scaled up learning with every additional worker, even though this was not as linear for multiple GPU devices as for parallel CPU threads. Finally, our end-to-end machine learning pipeline in SQL showed comparable performance to traditional machine learning frameworks.



(A) Linear regression (NY taxi).



(B) Neural network (MNIST).

FIGURE 6.12: End-to-end analysis of a machine learning pipeline: (a) linear regression (*New York taxi*, 64 tuples per batch and (b) a neural network (*MNIST*, stochastic gradient descent).

Chapter 7

ML2SQL: Compiling a Declarative Meta-Language for ML to SQL

Parts of this chapter have been published in [114, 115].

Database systems provide with SQL a declarative language that allows data manipulation and data retrieving without caring about optimisation details. With increasing hardware performance, database systems will not fully exploit the servers' hardware potentials as long as they are used for data retrieval only. To shift computation to the data stored in database systems, algorithms can be specified in SQL—as it has been Turing-complete since providing recursive tables—or as user-defined functions. The latter allows injecting code as stored procedures to be executed inside the database system and make an additional data manipulation layer on top obsolete. Even though the runtime would decrease, user-defined functions are not fully established as they form a mixture of declarative and procedural languages and are inconvenient to express for data scientists.

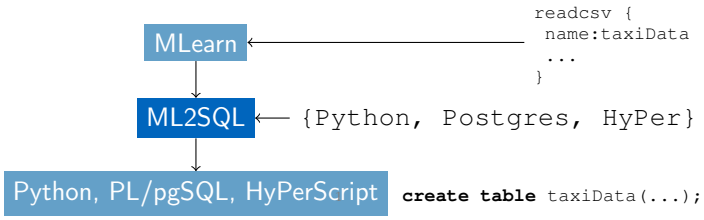


FIGURE 7.1: The conception of MLearn: as a meta-language, ML2SQL compiles code for a target (Python, PostgreSQL, HyPer), also including CSV file preprocessing.

When dealing with data and minimisation problems, dedicated tools as TensorFlow [1] or Pytorch form the status quo for performing machine learning tasks with tensors and gradient descent. Another approach of formulating machine learning tasks is using a declarative language as MLog [85], that compiles to code using TensorFlow, but, so far, it lacks support for use together with database systems. For computations inside database systems, the support of linear algebra together with matrices or tensors is essential. Different studies focus on the integration of linear algebra [91] and matrices inside database systems [71]. Going one step further, so-called array database systems replace relations by arrays as the native way of storing attributes. To support machine learning, TensorDB [73] aims at providing tensor calculus on top of array database systems. One study even provide its own declarative language (BUDS) [44] on top of a prototyped database system that supports matrix data types. Comparable domain-specific languages are Weld[1] for data driven workloads and IBM SystemML[2] [18] for creating flexible algorithms, but both cannot be used inside database systems. The intermediate language Ferry [48] allows translating from various code (i.e. Ruby or Haskell) into SQL but is not designed for use with array data types.

However, while linear algebra in database systems has been integrated and declarative language concepts have been proposed, there is no successful study on bringing a declarative language, tensor calculus and gradient descent in database systems together. Therefore,

[1] https://github.com/weld-project/weld
[2] https://systemml.apache.org

we develop a declarative machine learning language aimed at optimising models for supervised machine learning and data analysis. Our *MLearn language* allows specifying tasks independently of the target language, changing the underlying engine and making it easy to compare runtimes and results of different underlying frameworks (see Figure 7.1). This demonstration presents the *ML2SQL* compiler in particular, which compiles code written in *MLearn* to SQL (for PostgreSQL or HyPer [69]) or to Python using the frameworks NumPy and TensorFlow (see Figure 7.2).

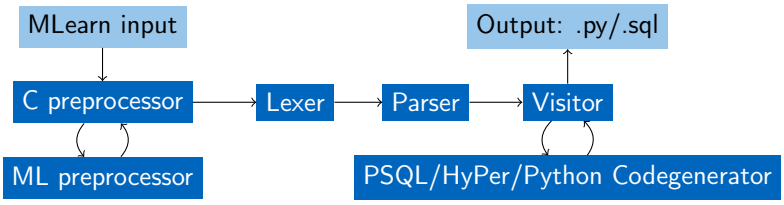


FIGURE 7.2: The compilation process: the *MLearn* language first gets preprocessed twice for handling includes, then the language gets tokenised and parsed. For each target platform, a generator allows translating the abstract syntax tree into the target language.

This chapter first introduces the *MLearn* language specifications and the details of the corresponding compiler. We evaluate the runtime of the generated code on the target platforms using the Chicago taxi dataset as input data and linear regression as optimisation model (optimised by gradient descent or as a closed-form solution). At the end, we introduce the demonstration concept including our web interface for online testing and conclude by improvements that might increase database systems' computation performance.

7.1 MLearn and the ML2SQL Compiler

The *MLearn* language is designed to define machine learning tasks in a declarative manner to be compiled to SQL or Python. It is intended as a meta-language to facilitate the use of database scripting languages for Python users. The ML2SQL compiler[3], implemented by Bungeroth [22], produces Python or SQL code runnable in PostgreSQL or HyPer. The compiler is written in ANTLR [101] and allows `import` and `include` statements. As the other computations rely on the initial data loading, the modular language design will allow the inclusion of `C` preprocessor statements (internally, it uses the `gcc -E` command). We allow two kinds of import statements, `include` and `import`. The first command allows basic text insertions known from the `gcc` preprocessor to load self-defined modules. The second command imports predefined libraries for time measurements, distributions and other convenient functions.

This section begins by introducing the language specification. We precede by listing the prerequisites of the target platforms in order to run the introduced tasks. Finally, we will give examples of how to use the *MLearn* language.

7.1.1 Language Specification

All operations work on integers, floating-point numbers, Boolean values or strings as basic types, which can be composed to tensors. On these types, *MLearn* provides the following features (see Listing 7.1):

- **Reading CSV files** as the fundamental operation to store the data in variables or relations of the database system.
- **Mathematical expressions** as provided by NumPy or SQL (as part of the projection operator).
- **Tensors** form the main part in our computations. Beside mathematical operations, we support accessing, slicing, concatenation and transposition.

[3] https://gitlab.db.in.tum.de/ml2sql/ml2sql

- **Functions** allow structuring the code and reducing code duplication. Also, external functions imported from other files or of a target language are allowed.
- **Control blocks.** In addition to our declarative statements, we allow conditional expressions and loops.
- **Distributions** are used for data sampling when initialising tensors with random values.
- **Preprocessor statements** as known from C can be used to include files and to allow the abstraction of different functions to different files.
- **k-fold cross-validation** as a predefined building block splits up a dataset into training and test sets to find the best—so-called—hyper-parameters.
- **Gradient descent** as a separate building block optimises the weights for a given loss function on input data.

```
expression = INJECT | 'print' mathexplist | 'if' '(' mathexp ')' '{' explist '}' ['else' '{' explist '}'] | ('continue' | 'break')
 | 'while' '(' mathexp ')' '{' explist '}' | 'for' VARNAME ('from' mathexp 'to' mathexp | 'in' interval) '{' explist '}' | functions
 | 'create' 'tensor' VARNAME 'from' VARNAME '(' varlist ')'
 | 'save' 'tensor' VARNAME 'to' (VARNAME|STRING) [':' STRING] '(' varlist ')'
 | VARNAME '[' accessor (',' accessor)* ']' ('=' | ':') mathexp | VARNAME (',' VARNAME)* '=' VARNAME '(' [mathexp (',' mathexp)*] ')'
 | VARNAME '[' accessor (',' accessor)* ']' '~' VARNAME '(' [mathexp (',' mathexp)*] ')' | 'import' VARNAME
 | VARNAME '=' (mathexp | 'distribution' '(' VARNAME ',' VARNAME ')') | VARNAME '~' VARNAME '(' [mathexp (',' mathexp)*] ')'
 | returnType* 'function' VARNAME '(' [VARNAME (',' VARNAME)*] ')' '{' explist '}' | 'return' mathexp (',' mathexp)*
 | ('readcsv'|'writecsv') '{' (('name:' VARNAME) | ('file:' STRING) | ('columns:' varlist) | ('replace_empty_entries:' mathexp)
                          |('delimiter:' STRING) | ('replace:' '{' (STRING ':' STRING)+ '}') | ('delete_empty_entries'))+ '}'
 | 'gradientdescent' '{' (('function:' STRING) | ('data:' varlist) | ('optimize:' [nameshape (',' nameshape)*])
                        | ('learningrate:' mathexp) | ('maxsteps:' mathexp) | ('batchsize:' mathexp) | ('threshold:' mathexp))+ '}'
 | 'plot''{'(('xData:'mathexp) | ('yData:'mathexp) | ('xLabel:'STRING) | ('yLabel:'STRING) | ('type:'STRING) |('filename:'STRING))+'}'
 | 'crossvalidate' '{' (('minfun' ':' VARNAME '=' fun) | ('kernel' ':' VARNAME ':' VARNAME ':' mathexp)
                      | ('data' ':' VARNAME (',' VARNAME)*) | ('n' ':' mathexp) | ('lossfun' ':' fun)
                      | ('folds' ':' mathexp) | ('test' '{' (VARNAME '=' interval)+ '}'))+ '}' ;
```

LISTING 7.1: Grammar of the MLearn language: Predefined building blocks for gradient descent, cross-validation, CSV file handling as well as procedural control blocks, function calls and the declaration of variables are allowed as expressions.

TABLE 7.1: Required matrix operations by MLearn.

Operation	Symbol	Arguments	Result
scalar mul.	$a * b$	$a \in \mathbb{R}, b \in \mathbb{R}^{n \times m}$	$\mathbb{R}^{n \times m}$
tensor mul.	$a * b$	$a \in \mathbb{R}^{n \times o}, b \in \mathbb{R}^{o \times m}$	$\mathbb{R}^{n \times m}$
tensor add.	$a + b$	$a, b \in \mathbb{R}^{n \times m}$	$\mathbb{R}^{n \times m}$
tensor sub.	$a - b$	$a, b \in \mathbb{R}^{n \times m}$	$\mathbb{R}^{n \times m}$
tensor power	a^b	$a \in \mathbb{R}^{n \times n}, b \in \mathbb{Z}$	$\mathbb{R}^{n \times n}$
transpose	a^t	$a \in \mathbb{R}^{n \times m}$	$\mathbb{R}^{m \times n}$
identity	$id(n)$	$n \in \mathbb{N}$	$\mathbb{B}^{n \times n}$
array fill	$fill(r, n, m)$	$r \in \mathbb{R}, n, m \in \mathbb{N}$	$\mathbb{R}^{n \times m}$

7.1.2 Target Language

We designed our machine learning language to compile to Python code with the libraries that data scientists would use. To work with SQL we picked out the disk-based database system PostgreSQL and its main-memory counterpart, HyPer. We assume for both systems an underlying scripting language, PL/pgSQL in PostgreSQL and HyPerScript in HyPer, that combines declarative SQL statements with procedural control blocks (see Chapter 4). The tensor operations in Python are performed using NumPy library calls. HyPer has already implemented all basic tensor operations including addition, (scalar) multiplication, power (including inverse for matrices on negative exponents), transposition, initialising an identity matrix and filling a matrix by a predefined value (see Chapter 3). As those operations do not exist in PostgreSQL (see Table 7.1), Bungeroth [22] has implemented these operations as C library function calls to the OpenBLAS library[4], also supporting parallelism. Furthermore,

[4] https://www.openblas.net/

we make use of predefined array operations as slicing and concatenation to divide the input dataset into training and testing one. Also, we wrap a PostgreSQL library extension around a gradient descent [124] library to allow in-database gradient descent in PostgreSQL.

7.1.3 Example

As an example, we specify linear regression as closed-form solution in *MLearn*:

$$\vec{w} = (X'^T X')^{-1} X'^T \vec{y}.$$

The example code (see Listing 7.2) splits a tensor (`A`) into features (`X`, the first three attributes, line `2`) and labels (`y`, line `3`). Then a tensor out of the value 1 as bias is prepended in front of the features (line `4/5`). Afterwards, the optimal weights (`w`) are computed out of tensor algebra (line `7`). The compiled code to Python can be seen in Listing 7.3, the one for PostgreSQL in Listing 7.5 and the one for HyPer in Listing 7.4. We can see that the code written in our declarative language is much more compressed.

```
1 A = [[1.1,0.98,87.3,3],[0.1,3.15,42.05,3.3],[100.5,26.8,10.1,225.1],[1097.5,23000,10.1,24850.1]]
2 X = A[: , 0:2]
3 y = A[: , 3]
4 bias[1,len(X,0)] : 1
5 X = (bias::X.T).T
6 Xt = X.T
7 w = (Xt*X)^(-1) * Xt * y
8 print '%' , w
```

LISTING 7.2: The specification in *MLearn*.

```
1  import numpy as np
2  def main():
3      A = np.array([np.array([1.1,0.98,87.3,3]),np.array([0.1,3.15,42.05,3.3]),np.array
           ([100.5,26.8,10.1,225.1]),np.array([1097.5,23000,10.1,24850.1])])
4      X = DATA[ :,0:2 +1]
5      y = DATA[ :,3:3 +1]
6      bias = np.full( ( 1,np.size(X ,0 )), 1)
7      X = (np.append(bias,X.T, axis=0)).T
8      Xt = X.T
9      w = np.dot( np.dot( np.linalg.matrix_power(( np.dot(Xt, X)), (-1)), Xt), y)
10     print( '{}'.format(  w))
11 if __name__ == "__main__": main()
```

LISTING 7.3: The translated code for Python using NumPy.

```
1  CREATE OR REPLACE FUNCTION ML_main() AS $$
2    var A = array[array[1.1::float,0.98::float,87.3::float,3],array[ 0.1::float,3.15::float
       ,42.05::float,3.3::float],array[100.5::float,26.8::float,10.1::float,225.1::float],array
       [1097.5::float,23000,10.1::float,24850.1::float]];
3    var X = array_resetlower(array_slice(A,1,array_length(A,1),(0+1)::int,(2+1)::int));
4    var y = array_resetlower(array_slice(A,1,array_length(A,1),(3+1)::int,(3+1)::int));
5    var bias = array_fill(1::float, 1,array_length(X,0+1));
6    X = array_transpose((array_cat(bias,array_transpose(X))));
7    var Xt = array_transpose(X);
8    var w = power((Xt*X), (-1)::int)*Xt*y; debug_print( '%',w);
9  $$ LANGUAGE 'hyperscript' strict;
10 select ML_main(); DROP FUNCTION ML_main();
```

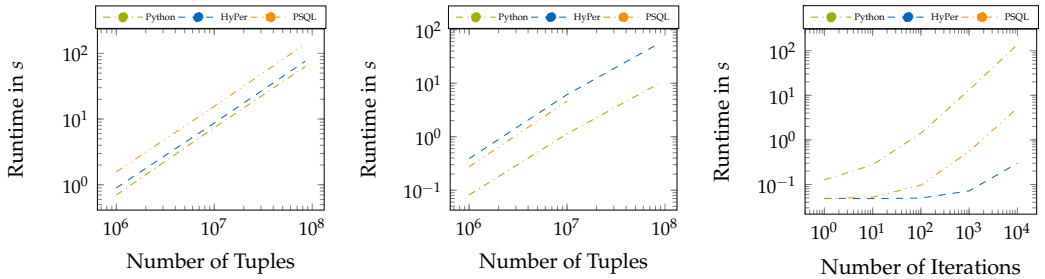
LISTING 7.4: As HyPerScript code for HyPer.

```
1  DO $$ declare
2    A float[][]; X float[][]; Xt float[][]; bias float[][]; w float[][]; y float[][];
3  begin
4    A := array[array[1.1::float,0.98::float,87.3::float,3],array[0.1::float,3.15::float,42.05::
       float,3.3::float],array[100.5::float,26.8::float,10.1::float,225.1::float],array[1097.5::
       float,23000,10.1::float,24850.1::float]]
5    X := A[:][0+1:2+1]; y := A[:][3+1:3+1];
6    bias := array_fill(1::float, ARRAY[1,array_length(X,0+1)]);
7    X := matrix_transpose((array_cat(bias,matrix_transpose(X))));
8    Xt := matrix_transpose(X);
9    w := matrix_power((Xt * X), (-1)::int ) * Xt * y;
10   RAISE NOTICE '%',w;
11 END$$;
```

LISTING 7.5: For PostgreSQL as PL/pgSQL procedure.

7.2 Evaluation

For evaluation (see Figure 7.3), we specify linear regression as closed-form or using gradient descent in our machine learning language and let the tasks run on the following target platforms: PostgreSQL version 10.5, Python 2.7.15 with NumPy 1.13.3 and TensorFlow 1.3.0 and the current HyPer system. We used an Ubuntu 18.04.01 LTS server with two sockets and twenty cores of Intel Xeon E5-2660 v2 processors in total (supporting hyper-threading). The server has 256 GiB of main-memory and uses 1 TiB of SSD as background storage. As test data served 85 mio. tuples of the Chicago taxi rides dataset[5].



(A) Loading data from CSV. (B) Linear regression: closed-form. (C) With gradient descent.

FIGURE 7.3: Runtime of (a) data loading from CSV, (b) solving linear regression using equation systems or (c) by gradient descent: For data loading and closed-form linear regression, we varied the input size; for gradient descent we varied the number of iterations. PostgreSQL did not support the needed array operations for more than 10^7 tuples.

We tested the runtime of loading data from CSV (see Figure 7.3a), the runtime of linear regression in closed-form (see Figure 7.3b) and by using gradient descent (see Figure 7.3c). For gradient descent, we used a learning rate of $5 \cdot 10^{-7}$ and varied the number of iterations from 1 to 10^4, but we used a constant input size (the whole dataset). The time measurements consider only the runtime needed for the specific operations, the time for data loading and array creation is measured separately.

The results show that data loading took in all three systems about the same time, no system seems to dominate one another. For the matrix operations used for closed-form linear regression, Python using NumPy still dominates the other systems (even PostgreSQL does not support two-dimensional array creation for more than 10^7 tuples). Hence, the integration of matrix calculus in database systems still has to be improved. Whereas using gradient descent, both database systems show competitive performance. Even some performance benefits originate from the used gradient descent optimiser, the results underline the possibility of analysing data inside the database system where it is stored.

In summary, the evaluation underlines the claim that the *ML2SQL* compiler makes it easy to compare different systems and that database systems show competitive performance on certain tasks.

7.3 Demonstration

For demonstration purposes, we have created an interactive web interface[6] (see Figure 7.4) that allows formulating tasks in *MLearn*, compiling to the choosable target language and executing the tasks. Switching between the different target platforms (Python, HyPer, PostgreSQL) makes it possible to compare the results and the runtimes of each target language. Behind the web interface, we run a PostgreSQL and a HyPer database server fed with an excerpt of the Chicago taxi dataset. The web interface allows everyone to try out the introduced types of tensor algebra as well as minimising arbitrary loss functions as, for example, linear or logistic regression.

[5] https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew/data
[6] http://ml2sql.db.in.tum.de

FIGURE 7.4: Web interface for an interactive exploration of the *MLearn* language: Above left, the text editor allows specifying tasks, which are translated into the selected target language (above right). The code will be executed in the terminal.

7.4 Conclusion

This chapter presented the first declarative machine learning language *MLearn*, which allows describing machine learning tasks independently of the target engine and whose compiler allows running the code in the core of database systems. This chapter first introduced comparable approaches before it presented the language specifications for performing linear regression and gradient descent using any possible loss function. Then, we evaluated the runtime of the tasks on the different target platforms PostgreSQL, HyPer and Python using NumPy and TensorFlow. The results showed, that it was indeed feasible to run the tasks as stored procedures inside database systems showing comparable runtime, especially during matrix creation.

Overall we have shown the potential of a declarative machine learning language of expressing tasks compactly and being independent of the underlying engine.

To sum up, we have discovered out that array processing represents the major building block for tasks related to machine learning. These tasks would strongly benefit from SQL especially for data preprocessing. As future work—to boost the capabilities of database systems for array data—remains the development of efficient array data types and the standardised integration of optimisation methods such as gradient descent inside database systems.

In addition, when integrating the advantages of array databases into hybrid OLTP and OLAP database systems, no domain-specific systems would be required. We shall therefore work on applying matrix algebra to tables using stored procedures that are written in ArrayQL.

Chapter 8

ArrayQL

Parts of this chapter have been published in [116, 117].

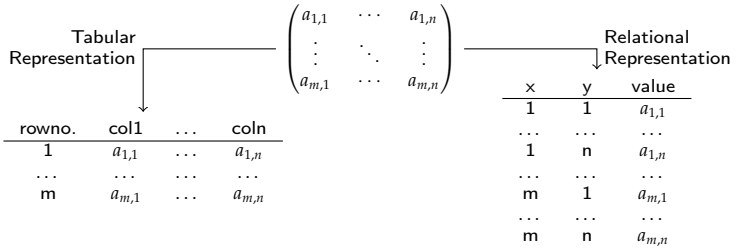


FIGURE 8.1: Storing arrays as database tables: either using a tabular representation (left) with the attributes as columns and an additional one defining the row order or using a relational representation (right) with the array as coordinate list.

Array database systems are developed for geo-temporal data and therefore specialised for multidimensional discrete data (MDD) [6]. Such data occurs within time-series of scientific experiments or real-world scenarios when processing images or indexing geographic or astronomical data. These examples have in common, that the tuples can be addressed using a multi-dimensional index out of coordinates and time. In contrast to relational database systems, array database systems are designed for index-based array access and excel in computing aggregations on numerical data. Popular array database systems are RasDaMan [6], MonetDB SciQL [147] and SciDB [38, 34]. As each one is shipped with its own query language, ArrayQL [87] is an attempt to standardise them that has been presented at XLDB 2012. Although the corresponding algebra [94] has been published, it is not fully covered by the corresponding draft of a grammar specification [87], which is needed in order to implement ArrayQL.

Even though array database systems are often based on relational ones, an interface for querying both does not exist. For example, RasDaMan supports relational database systems such as PostgreSQL as an underlying key-value store but archives the data as binary large objects (BLOB) only. SciQL is implemented within MonetDB and stores arrays along with tables in the same memory layout but does not enable cross-querying. But to allow access from SQL *and* an array query language, a uniform representation is needed. Arrays have to be either stored as coordinate list (*relational representation*) or tables have to carry an additional attribute that defines the row order (*tabular representation*, see Figure 8.1). A relational representation saves memory on sparse arrays as no entry is needed for values equal to zero. Also primitive data types as attributes are sufficient even for more than two dimensions. A tabular representation would require an array data type to represent the third dimension.

Another use case for array-oriented data processing arises by the need of matrix operations for data mining and machine learning [146, 145, 133]. The corresponding data is often stored and collected inside of relational database systems, but its analysis depends on linear algebra, which database systems do not provide natively. Thus, the data gets extracted into separate tools such as R and Python, so analysis happens on past data, ignoring incoming tuples. We argue that array database systems are ideally suited for machine learning algorithms, which essentially depend on data stored in tensors and their transformations, making ArrayQL a worthwhile extension.

We claim that relational database systems will highly benefit from ArrayQL as a further query language, either embedded in SQL as user-defined functions or as a separate query interface.

We integrate ArrayQL within our code-generating database system Umbra. We decide in favour of a relational array representation, which allows a direct mapping onto relational algebra at compile time. This requires an extension of the semantic analysis only, rather than a change to the underlying query engine. The extension accepts ArrayQL statements as part of SQL as user-defined functions or via a separate interface. As an advantage, ArrayQL can work on SQL tables and SQL has access to ArrayQL arrays without modification and the extension does neither affect runtime nor the compile time of SQL queries. This chapter provides a translation of the so far known ArrayQL operators into relational algebra and a corresponding grammar. This chapter's specific contributions are:

- an extended grammar definition that supports the full ArrayQL algebra,
- a relational array representation including bounding boxes and validity maps for ArrayQL within database systems
- the translation of corresponding operators into relational algebra,
- the integration of ArrayQL into a code-generating database system with Umbra as target,
- and an experimental evaluation using micro-benchmarks for linear algebra and real-world data for array operations.

This chapter comprises the following sections: Section 8.1 presents the architecture when integrating ArrayQL within the beyond main-memory database system Umbra as the target. Section 8.2 introduces the ArrayQL algebra and operator translation to relational algebra. Section 8.3 provides a complete ArrayQL grammar to address the presented ArrayQL operators. Section 8.4 demonstrates the application of ArrayQL in conjunction with SQL and for linear algebra. Section 8.5 evaluates the proposed extension using micro-benchmarks for basic matrix algebra, queries on real-world data and the SS-DB benchmark.

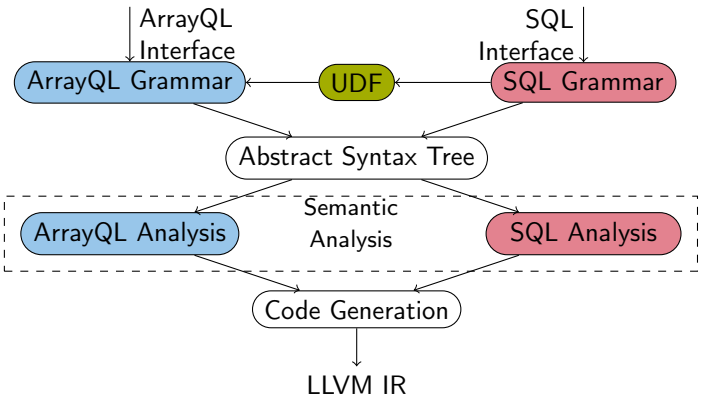


FIGURE 8.2: ArrayQL extension in Umbra: ArrayQL statements are parsed via a separate interface or as part of user-defined functions (UDF) in SQL; a separate file contains the ArrayQL grammar. An abstract syntax tree is generated, whereof the ArrayQL statements are analysed separately.

8.1 In-Database Integration

This section illustrates the changes made to the database system Umbra either in order to accept ArrayQL as user-defined functions or to offer a separate query interface. Afterwards, we explain design decisions made to suit the relational concept of a code-generating database system and to enable cross-querying without overhead.

8.1.1 Architecture

Umbra is a code-generating database system following the producer-consumer concept. First, it transforms the parsed SQL query into an abstract syntax tree, which is passed to the semantic analysis to create the operator plan. The operator plan gets optimised using estimated cardinalities as only the schema is known during compile time. Afterwards, a translator class for each operator generates the LLVM code, for which the traditional tuple-flow is inversed. So instead of pushing tuples upwards a target operator, operators demand their sources to produce code. Each source then demands the parent operator, the consumer, to generate code for processing each tuple further.

Umbra supports user-defined functions, that are handled separately during semantic analysis. For each language, a separate grammar file together with one for its semantic analysis exists. This is shown in Figure 8.2: when adding ArrayQL to Umbra, either a separate interface accepts ArrayQL statements or they are parsed as part of a user-defined function in SQL. Afterwards, a common abstract syntax tree is generated, which is then analysed separately. Within the semantic analysis, we mostly rely on standard relational algebra operators, but we are also able to call customised operators. Because of this, we benefit from query optimisation such as operator reordering or predicate push-down.

8.1.2 Array Representation

Only the schema is known during compile time, whereas the tuples can only be accessed during runtime. This interferes with a tabular array representation, as only the columns are part of the schema, and leads us to the relational representation. We store every n-dimensional array with m values per cell as a table with $n + m$ attributes. Stored as a coordinate list, the attributes for the indices are unique and form the primary key. This allows their indexing and fast retrieval later on.

ArrayQL differentiates between attributes and dimensions, which becomes obsolete in a relational representation as dimensions are mapped to attributes internally. This leads to more flexibility, since arbitrary attributes can be used as dimensions.

According to the ArrayQL algebra, an array consists of a *bounding box*, a *validity map* and the *content*. The bounding box defines the bounds for each dimension, whereas the validity map defines the visible cells within the bounds and the attributes per cell define the content. To define the bounding box, we simply insert a tuple for the lower as well as the upper bound upon array creation (see Figure 8.3). Within the bounding box, we consider an entry as valid if it exists and at least one attribute is not declared as `NULL`.

```
1 CREATE ARRAY m (
2  i INTEGER DIMENSION [1:2],
3  j INTEGER DIMENSION [3:4],
4  v INTEGER);
```

→

x	y	v
1	3	*NULL*
2	4	*NULL*

FIGURE 8.3: When an array is created with ArrayQL and the dimensions are specified, a corresponding relation is created with two initial tuples defining the bounding box.

8.1.3 Interaction with SQL

Depending on its signature, ArrayQL expressions, when used as part of a user-defined function, return either a table, e.g., `TABLE (x INT, y INT, v INT)`, or a single array attribute, e.g., `INT[][]` (see Listing 8.1). As a table function, it returns the relational array representation, that can be further processed in SQL. Otherwise, when the function is declared to return a single attribute, the result is cast to Umbra's array data type.

```
1 CREATE FUNCTION exampletable() RETURNS TABLE (x INT, y INT, v INT) LANGUAGE 'arrayql' AS 'SELECT
      [x],[y],v FROM m';
2 CREATE FUNCTION exampleattribute() RETURNS INT[][] LANGUAGE 'arrayql' AS 'SELECT [x],[y],v
      FROM m';
```

LISTING 8.1: ArrayQL as part of a user-defined function returns either an SQL table or an SQL array.

TABLE 8.1: Operators of the ArrayQL algebra: the first column names the operator, the second column specifies the input arguments, the third column the output array, the fourth column defines the set of valid indices and the latter one the translation of ArrayQL operators into relational algebra. $i_{1...n}$ represents the attribute for the dimension in relational form, $|i_{1...n}|$ denotes the size of a dimension. We assume arrays having a single attribute $v \in \mathbb{R}$ only.

Operator	Input	Output	Validity Map	Relational Algebra
apply	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}, f \in (\mathbb{R} \to \mathbb{R})$	$\mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}$	$d_a = d_{out} \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert$	$\pi_{i_1,\ldots i_n, f(v)}(a)$
combine	$\mathfrak{a}, \mathfrak{b} \in \mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}$	$\mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}$	$d_a \uplus d_b = d_{out} \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert$	$a \overset{\cdot}{\bowtie}_{a.i_1=b.i_1 \wedge \cdots \wedge a.i_n=b.i_n} (b)$
i. d. join	$\mathfrak{a}, \mathfrak{b} \in \mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}$	$(\mathbb{R}, \mathbb{R})^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}$	$d_a \cap d_b = d_{out} \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert$	$a \bowtie_{a.i_1=b.i_1 \wedge \cdots \wedge a.i_n=b.i_n} (b)$
fill	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}$	$\mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}$	$d_a \subseteq d_{out} = \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert$	$\ldots 0_{\lvert a.i_1\rvert,\ldots,\lvert a.i_n\rvert} \cdots$
filter	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}, p \in (\mathbb{R} \to \mathbb{B})$	$\mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}$	$d_{out} \subseteq d_a \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert$	$\sigma_{p(v)}(a)$
rebox	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}, i_1^l, i_1^u, \ldots, i_n^l, i_n^u \in \mathbb{I}$	$\mathbb{R}^{i_1^u - i_1^l \times \cdots \times i_n^u - i_n^l}$	$d_a \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert, d_{out} \subseteq i_1^u - i_1^l \times \cdots \times i_n^u - i_n^l$	$\sigma_{i_1^l \le i_1 \le i_1^u \wedge \cdots \wedge i_n^l \le i_n \le i_n^u}(a)$
reduce	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}, f \in (\mathbb{R}^{\lvert i_n\rvert} \to \mathbb{R})$	$\mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_{n-1}\rvert}$	$d_a \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert, d_{out} \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_{n-1}\rvert$	$\gamma_{i_1,\ldots,i_n,f(v)}(a)$
rename	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}$	$\mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}$	$d_a = d_{out} \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert$	$\rho(a)$
shift	$\mathfrak{a} \in \mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}, i'_1, \ldots, i'_n \in \mathbb{I}$	$\mathbb{R}^{\lvert i_1\rvert \times \cdots \times \lvert i_n\rvert}$	$d_a = d_{out} \subseteq \lvert i_1\rvert \times \cdots \times \lvert i_n\rvert$	$\pi_{i_1+i'_m,\ldots,i_n+i'_n,v}(a)$

8.2 ArrayQL Algebra

ArrayQL offers an algebra [94] that is similar to relational algebra and allows a mapping to SQL operators considering the underlying schema. The algebra offers nine operators (see Table 8.1), for which it defines content, validity maps and bounding box. In our relational form, one relation $a \subseteq \mathbb{I}^n \times \mathbb{R}^m$ with schema $sch(a) = \{\underline{i_1, \ldots, i_n}, r_1, \ldots, r_m\}$ represents one n-dimensional array $\mathfrak{a} \in (\mathbb{R}^m)^{\lvert i_1\rvert \times \ldots \times \lvert i_n\rvert}$ with m attributes of domain $\mathbb{R}$ as content. Its coordinates $(i_1, \ldots, i_n) \subseteq \mathbb{I}^n$ form the primary key and delimit the bounding box. We formulate the validity map of an array $\mathfrak{a}$ as set of indices $d_a \subseteq \mathbb{I}^n$ of valid entries. Transferred to SQL, all entries are valid, for which a tuple exists with not-null attributes. This section introduces the ArrayQL operators, the corresponding syntax and the translation into SQL operators.

8.2.1 Rename

The rename operator assigns a new name to either a dimension, attribute or a whole array. Similar to the rename operator ρ in SQL, it is introduced by a keyword (`AS`) behind expressions or tables.

```
SELECT [i] AS s, [j] AS t, v AS c FROM m[s,t];
```

LISTING 8.2: Rename operator.

8.2.2 Function Application

The apply operator applies a function $f \in \mathbb{R}^m \to \mathbb{R}^o$ on certain attributes of each valid entry. This is translated to an arithmetic expression as part of an SQL projection $\pi_{i_1,\ldots,i_n,f(r_1,\ldots,r_m)}(a)$. As function application does not affect the validity map, no further adjustments are needed.

```
SELECT [i], [j], v+2 FROM m;
```

LISTING 8.3: Function application: addition.

8.2.3 Filter

The filter operator invalidates cells for which a condition does not hold. This is called implicitly when accessing an array via indices or explicitly when checking the cell's value as part of the `WHERE`-clause. Both ways are translated into selections of relational algebra $\sigma_{p(v)}(a)$, as both dimensions and attributes are represented in SQL as attributes.

```
SELECT [i], [j], v FROM m WHERE v = 0.0;
SELECT [i] as i, [j] as j, * FROM m[i/2, j];
```

LISTING 8.4: Explicit and implicit filter operator.

8.2.4 Index Manipulation: Shift and Rebox

Shift moves the indices, whereas rebox redefines the bounding boxes by enlarging or shrinking the array size. In our relational schema, shift is translated into an arithmetic expression as part of a projection, as it modifies each index by adding or subtracting the difference $i'_1, \ldots, i'_n \in \mathbb{I}$:

$$\pi_{i_1+i'_1,\ldots,i_n+i'_n,r_1,\ldots,r_m}(a).$$

```
SELECT [i] as i, [j] as j, b FROM m[i+1,j-1];
```

LISTING 8.5: Shift operator.

For rebox, if the array size is shrunk, a conditional statement (selection) filters out each index, which is outside the new bounding box given as lower and upper bounds $i^l_1, i^u_1, \ldots, i^l_n, i^u_n \in \mathbb{I}$:

$$\sigma_{i^l_1 \leq i_1 \leq i^u_1 \wedge \cdots \wedge i^l_n \leq i_n \leq i^u_n}(a).$$

In any case, new array bounds have to be added afterwards (with a union operator).

```
SELECT [1:5] as i, [1:5] as j, * FROM m[i,j];
```

LISTING 8.6: Rebox operator.

8.2.5 Fill

The fill operator creates an entry with the default value (0 for numerics) for the attributes of every invalid cell within the bounding box. This is useful for linear algebra with arrays as input matrices and has to be called by a keyword. Internally, it is translated to a call to `generate_series`, an outer join and a projection, only when enabled by the keyword `filled` in ArrayQL and needed before applying specific operations (see Section 8.4.2):

$$\pi_{COALESCE(a.r_1,0),\ldots}(a \rtimes\!\ltimes_{a.i_1=b.i_1 \wedge \cdots \wedge a.i_n=b.i_n} (\rho_b(0_{|a.i_1|,\ldots,|a.i_n|}))).$$

```
SELECT FILLED [i], [j], * FROM m;
```

LISTING 8.7: The keyword `FILLED` enables the fill operator.

8.2.6 Combining and Joining

ArrayQL defines three operators for joining arrays, namely combine, the inner dimension join and—its generalisation to attributes—the inner extended join.

Combine

Combine merges two arrays of the same dimensionality but distinct valid cells, so it concatenates arrays. All cells are valid that are at least valid in one input: $d_a \uplus d_b = d_{out} \subseteq |i_1| \times \cdots \times |i_n|$. `NULL` is assumed for the attributes of a missing join partner. Combine acts like a full outer join, to which it is translated in relational algebra:

$$a \rtimes\!\ltimes_{a.i_1=b.i_1 \wedge \cdots \wedge a.i_n=b.i_n} (b).$$

```
CREATE ARRAY m2(x INTEGER DIMENSION [3:4], y INTEGER DIMENSION [1:2], v2 INTEGER);
SELECT [i] as i, [j] as j, v, v2 FROM m[i,j], m2[i,j];
```

LISTING 8.8: Combine operator.

Inner Join

The inner dimension/extended join corresponds to the inner join:

$$a \bowtie_{a.i_1=b.i_1 \wedge \cdots \wedge a.i_n=b.i_n} (b).$$

All cells are valid, that are valid in both join partners: $d_a \cap d_b = d_{out} \subseteq |i_1| \times \cdots \times |i_n|$. They differ, as the inner dimension join only allows dimensions as indices, whereas the inner extended join generalises the join predicate, so that attributes can be used to determine the index as well. As the usage of either combine or join is data-dependent and not known during compile time, we add the keyword `JOIN` to explicitly perform an inner join. This differs from the original ArrayQL proposal where it shares the syntax with combine (which is called when an inner join cannot be applied).

```
1 SELECT [i] as i, [j] as j, v, v2 FROM m[i+2,j+2] JOIN m2[i-2,j-2];
```

LISTING 8.9: Inner dimension Join.

8.2.7 Reduce for Aggregations

Reduce performs an aggregation over at least one dimension as needed by roll-up queries of analytical workloads. Reduce is introduced by the keywords *GROUP BY*, as known from SQL, followed by the preserved dimensions after reduction. Similarly, one aggregation function $f \in ((\mathbb{R}^m)^{|i_n|} \to \mathbb{R}^m)$ must be applied to all remaining attributes. These similarities allow a direct mapping to aggregations in relational algebra:

$$\gamma_{i_1,\ldots,i_{n-1},f(v)}(a).$$

```
1 SELECT [i], sum(v) FROM m GROUP BY i;
```

LISTING 8.10: Reduce operator for aggregation: summation

8.3 ArrayQL Grammar

The syntax draft of 2012 [87] proposed ArrayQL as a data definition and data query language on arrays as the principle data model. Common elements with SQL are the keywords and the syntax to create and access attributes, but in contrast, the statements are extended to specify dimensions. Beside arithmetic operations and basic array transformations, ArrayQL allows aggregations and joins. We add support for temporal tables, extend the join functionality and propose the syntax for an extension to a data modification language. In this section, we introduce the ArrayQL statements and give a syntax definition in extended Backus-Naur-Form (see Listing 8.11).

8.3.1 Data Definition Language

As a data definition language, ArrayQL allows the creation of arrays similar to SQL tables (see Listing 8.12). A create statement starts with the keyword `CREATE ARRAY` followed by the array name. As arguments inside parentheses, ArrayQL expects the keyword `DIMENSION` together with the array bounds and, as in SQL, the attributes per cell.

```
1 CREATE ARRAY m (i INTEGER DIMENSION [1:2], j INTEGER DIMENSION [1:2], v INTEGER);
```

LISTING 8.12: Array creation statement.

As an alternative, creation is possible out of an existing array (see Listing 8.13).

```
1 CREATE ARRAY n FROM SELECT [i], [j], v FROM m;
```

LISTING 8.13: Array creation out of existing array.

```
Statement  = ( ( CreateStmt | SelectStmt | UpdateStmt ) ";" )*;

CreateStmt = "create" "array" NAME ( ("from" SelectStmt) | ArrayDef) ;
ArrayDef   = "(" Dimension ("," Dimension)* "," Attribute ("," Attribute)* ")";
Dimension  = NAME Type "[" INT (":" INT)* "]";
Attribute  = NAME Type ("default" EXPR)?;
Type       = "integer" | "float" | "text";

SelectStmt = "with array" NAME "as" (("from" SelectStmt) | ArrayDef)
                    ("," NAME "as" (("from" SelectStmt) | ArrayDef))
             "select" SelectExpr ("," SelectExpr)* "from" SubArray ( ( "," | "join" SubArray)*
             ("where" BOOL)? ("group by" NAME ("," NAME)* )?;
SubArray   = NAME  ("["  EXPR  ("," EXPR )* "]")?  NAME? |  NAME "*" NAME ;
SelectExpr = EXPR ("as" NAME)? | "[" NAME "]" ( "as" NAME )? | "[" INT ":" INT "]" "as" NAME
           | ("min" | "max" | "avg" | "sum" | "count") "(" NAME ")";

UpdateStmt = "update" (NAME "[" EXPR "]" | "[" INT ":" INT "]")* "(" UpExpr ")";
UpExpr     = SelectStmt | "values" "(" EXPR ( "," EXPR )* ")" ( "," "(" EXPR ( "," EXPR )* ")");
```

LISTING 8.11: Grammar of ArrayQL. `NAME`, `EXPR`, `BOOL`, `INT` denote a text string, arithmetic, Boolean or integer expression.

We suggest ArrayQL in conjunction with SQL insert statements to allow mixed queries. In conjunction with SQL, the ArrayQL create statement allows to create new tables or prepare existing ones for array-based processing (initialising the bounds and adding an index on the dimension attributes). When a new array has been created, SQL can access the corresponding table to insert elements like bulk-loading from CSV. Afterwards, ArrayQL as a data query language can process the filled array.

8.3.2 Data Query Language

ArrayQL is intended as a data query language to access and aggregate along the array's dimensions. Similar to SQL, an ArrayQL statement is composed out of a `SELECT`- and a `FROM`-clause and, optionally, one for `WHERE` and one for `GROUP BY` (see Listing 8.14). The `SELECT`-clause expects the indices for the dimensions (in brackets) as well as arithmetic expressions as attributes that form the result. The `FROM`-clause specifies the source array. As its arguments, arrays or compound statements, such as joins, table-functions or entire subqueries are allowed. The `WHERE`-clause filters each entry by a predicate. The `GROUP BY`-clause addresses the dimensions preserved after an aggregation.

```
1 SELECT [i], SUM(v)+1 FROM m WHERE v>0 GROUP BY i
```

LISTING 8.14: Exemplary ArrayQL select statement.

Optionally, the indices can be rearranged, which is indicated by the dimension names inside brackets behind the source array. The keyword `AS` allows renaming of expressions as well as of input arrays. Filtering and grouping happen similar to SQL: filter expects a condition inside a `WHERE`-clause, grouping expects the dimensions considered for aggregation as part of the `GROUP BY`-clause.

In addition to the ArrayQL draft, we propose support of temporary arrays, explicit inner joins and various table functions. Temporary arrays are introduced by the keyword `WITH` (see Listing 8.15), similar to temporary tables in SQL.

```
1 WITH ARRAY tmp AS (SELECT [i] as j, SUM(v+1) FROM m[j] WHERE v>0 GROUP BY j) SELECT * FROM tmp;
```

LISTING 8.15: Temporary table in ArrayQL.

Inner joins are part of the ArrayQL algebra in Section 8.2. Table functions are needed to apply linear algebra on arrays, and therefore, discussed in Section 8.4.

8.3.3 Data Modification Language

Beside manipulating existing arrays using SQL, we add basic support for ArrayQL update statements (see Listing 8.16). An ArrayQL update statement is introduced by `UPDATE ARRAY` and expects the array name and the dimensions whose values should be modified.

An ArrayQL select statement or an explicit `VALUES`-clause containing the attributes specify the new values. Inside a `VALUES`-clause all attributes of one cell are enclosed by brackets.

```
UPDATE ARRAY m [1][1] (select [2][1],v FROM m);
UPDATE ARRAY m [1][1:2] (values (1),(2));
```

LISTING 8.16: ArrayQL update statements.

8.4 Application of ArrayQL

Array query languages have two major application areas: the common one is for use with geo-temporal data, the second one is applying linear algebra on data in relational form for statistical analysis. One important difference between the two domains concerns the handling of invalid values. Array database systems assume `NULL` for non-existing values, whereas these are interpreted as `0` within sparse matrices. To conform to the ArrayQL specification, we assume the geo-temporal use-case as default. To distinguish the other one, the keyword `filled` indicates matrix operations. This section explains both use-cases including table-function extensions.

8.4.1 Geo-Temporal Data

The intended purpose of ArrayQL is to allow index-based access to geo-temporal data. With our ArrayQL integration into a relational database system, mixed query types become possible. A table created in SQL (see Listing 8.17) can be accessed within ArrayQL (see Listing 8.18) and vice versa. ArrayQL interprets the attributes that form the table's primary key as indices. Accordingly, SQL has access to all array's dimensions as attributes.

```
CREATE TABLE taxidata(id TEXT, pickup_longitude INT, pickup_latitude INT, pickup_datetime DATE,
    dropoff_datetime DATE, trip_duration FLOAT, PRIMARY KEY(id, pickup_longitude,
    pickup_latitude));
INSERT INTO mytaxidata [..];
```

LISTING 8.17: Table creation and data insertion using SQL.

```
SELECT [pickup_longitude],[pickup_latitude], SUM(trip_duration)
FROM mytaxidata GROUP BY pickup_longitude, pickup_latitude;
```

LISTING 8.18: ArrayQL queries on an SQL table: the attributes that form the primary key serve as indices.

8.4.2 Linear Algebra with ArrayQL

The ArrayQL operators allow expressing basic mathematical expressions. Complex functions are realised by dedicated operators or user-defined functions. They are called as part of the `from`-clause where arbitrary table functions are accepted as input.

When creating an array, the bounding box defines the size of a matrix, the attributes determine the field of which each entry is a member. Operations on matrices obey the following pattern: scalar operations are part of the `select`-clause, matrix operations belong to the `from`-clause (see Figure 8.4).

$$(a_{ij}) = A \in \mathbb{R}^{2\times 3}$$

```
CREATE ARRAY A (
i INTEGER DIMENSION [1:2],
j INTEGER DIMENSION [1:3],
a FLOAT)
```

$$(a_{ij}) = A$$

```
SELECT [i],[j], A.a FROM A
```

FIGURE 8.4: Correlation of ArrayQL with matrices.

TABLE 8.2: Matrix algebra with ArrayQL.

Function	ArrayQL operator
addition	apply
scalar multiplication	apply
matrix multiplication	i.d.join, reduce
slice	rebox
subtraction	apply
transpose	rename

When performing linear algebra on arrays, the main difference to geo-temporal applications concerns the meaning of invalid entries. As arrays are interpreted as sparse matrices, values of non-existing entries are assumed to be zero. The fill operator has to be put implicitly in front of respective operations to consider operations that alter zero values. To enable this feature, we propose the keyword `filled` behind `select`. Having enabled this feature, the fill operator presented in Section 8.2.5 is called when necessary: this includes all arithmetic unary and binary functions but not joins. This functionality is enabled for all trigonometric, arithmetic and aggregate functions (see Listing 8.19). During semantic analysis, the fill operator is added to the operator tree before the function call is generated. The operator creates an array of the same dimensions containing zeros proceeded by an outer join with the original table on the indices. A `COALESCE` statement then replaces non-existing (`NULL`) values, thus within the array-bounds only.

```
1 SELECT FILLED [i], [j], v+2 FROM m;
2 SELECT FILLED [i],  max(v) FROM m GROUP BY i;
```

LISTING 8.19: The fill operator is called before a function call, for example, of an arithmetic binary function or a unary aggregate function like a row-wise maximum function.

The ArrayQL algebra allows expressing basic mathematical expressions as scalar operations or compound statements (see Table 8.2). For operations not covered by the algebra, we add additional table functions. We demonstrate matrix operations expressed in ArrayQL, short-cuts to simplify their usage and their application for linear regression and training a neural network.

Scalar Operations

Scalar operations are part of the `select`-clause as arithmetic expressions (see Listing 8.20). They correspond to the apply operator, since scalar multiplication, addition or subtraction are each applied elementwise.

```
1 SELECT [i],[j],m.v*n.v FROM m,n;   -- multiplication
2 SELECT [i],[j],m.v+n.v FROM m,n;   -- addition
3 SELECT [i],[j],m.v-n.v FROM m,n;   -- subtraction
```

LISTING 8.20: Scalar operations in ArrayQL.

Transpose and Slice

To transpose or slice an array, we can rely on basic index manipulations. Slicing an array corresponds to the rebox operator (see Listing 8.6), as it defines a sub-range for each index. For transposition $A^T = (a_{ij})^T = (a_{ji})$, ArrayQL does not perform an operation but renames the indices (see Listing 8.21) as the matrices are stored in a relational representation as a coordinate list.

```
1 SELECT [j] AS s, [i] AS t, * FROM m[s, t]
```

LISTING 8.21: Transpose.

TABLE 8.3: Short-cuts for matrix operations (`from`-clause).

Function	Short-cut	Input	Output
addition	$\square + \square$	$M \times N, M \times N$	$M \times N$
inversion	$\square^{-1}$	$M \times N$	$M \times N$
multiplication	$\square * \square$	$M \times K, K \times N$	$M \times N$
power	$\square^{\square}$	$M \times M$	$M \times M$
subtraction	$\square - \square$	$M \times N, M \times N$	$M \times N$
transposition	$\square^{T}$	$M \times N$	$N \times M$

Matrix Multiplication

The text-book implementation of a matrix multiplication $(a_{i,k}) \cdot (b_{k,j}) = \sum_{k=1}^{n} a_{ik} b_{kj}$ can be expressed in ArrayQL. In relational algebra, with two tables $m\{[i,k,v]\}, n\{[k,j,v]\}$ each representing a matrix, the multiplication corresponds to an operator tree out of join, apply (for elementwise multiplication) and reduce (for final summation):

$$\gamma_{m.i,n.j,sum(m.v \cdot n.v)}(\pi_{m.v \cdot n.v}(m \bowtie_{m.k=n.k} n)).$$

ArrayQL allows to join over the dimension k implicitly (see Listing 8.22). Nevertheless, this query highly resembles its SQL counterpart (see Listing 8.23). In addition, it is not practical as the product and the summation have to be stated manually.

```
SELECT [i], [j], SUM(product) AS a FROM (
  SELECT [*:*] AS i, [*:*] AS j, [*:*] AS k, a.v * b.v AS product
  FROM m[i,k] a JOIN n[k,j] b) as ab GROUP BY i,j;
```

LISTING 8.22: Text-book matrix multiplication in ArrayQL.

```
SELECT m.j AS i,n.j,SUM(m.v*n.v)
FROM a AS m INNER JOIN a AS n ON m.i=n.i
GROUP BY m.j, n.j;
```

LISTING 8.23: Corresponding matrix multiplication in SQL.

Implemented Table Functions and Short-Cuts

To express linear algebra with ArrayQL, we introduce abbreviations for matrix operations. Matrix operations, either expressible in ArrayQL (like matrix multiplication or addition), or requiring a table function (such as for inversion) should belong to the `from`-clause. We implemented short-cuts to offer similar notations that resemble mathematical expressions and avoid writing prefix function calls (like `f()`). These short-cuts exist for operations not covered so far by the ArrayQL algebra as well as for compound operations to simplify their application (see Listing 8.24, Table 8.3). Furthermore, this allows the `from`-clause to comprise larger statements out of matrices later needed for linear regression.

```
SELECT [i],[j],* FROM m+n;  -- addition
SELECT [i],[j],* FROM m^-1; -- inversion
SELECT [i],[j],* FROM m*n;  -- multiplication
SELECT [i],[j],* FROM m^2;  -- power
SELECT [i],[j],* FROM m-n;  -- subtraction
SELECT [i],[j],* FROM m^T;  -- transposition
```

LISTING 8.24: Short-cuts in ArrayQL.

Application

The presented operations allow solving numerical tasks based on matrix manipulations like solving linear regression numerically or training a neural network.

Linear regression can be expressed out of an input array $X \in \mathbb{R}^{i \times j}$, containing i tuples with j attributes, and a weight vector $\vec{w} \in \mathbb{R}^j$ to predict the labels $\vec{y} \in \mathbb{R}^i$ as $X \cdot \vec{w} = \vec{y}$. Given

a training dataset with corresponding labels $\vec{y}$, a closed-form expression out of transposition, matrix multiplication and inversion computes the optimal weight matrix:

$$\vec{w} = (X'^T X')^{-1} X'^T \vec{y}.$$

As multiplication and transposition are expressible in ArrayQL, the corresponding shortcuts together with one for matrix inversion allow ArrayQL to process the closed-form expression. The query computes the weight matrix (see Listing 8.25) without writing nested subqueries as in SQL (see Listing 8.26).

```
SELECT [i],[j],* FROM ((m^T * m)^-1*m^T)*y
```

LISTING 8.25: Linear regression in ArrayQL.

```
SELECT tmp.i, SUM(tmp.sum*y.val) FROM (
 SELECT inv.i, m.i as j, sum(m.val*inv.sum)
 FROM matrixinversion(TABLE (
  SELECT a1.j AS i,a2.j,SUM(a1.val*a2.val)
  FROM m AS a1 INNER JOIN m AS a2 ON a1.i=a2.i
  GROUP BY a1.j, a2.j)
 ) AS inv INNER JOIN x ON inv.j=a.j GROUP BY inv.i, m.i
) AS tmp INNER JOIN y ON tmp.j=y.i GROUP BY tmp.i;
```

LISTING 8.26: Corresponding query in SQL.

Supporting transposition and multiplication on arrays, ArrayQL is capable of expressing the forward pass of a fully connected neural network. A fully connected neural network with one hidden layer of size h requires two weight matrices $w_{hx} \in \mathbb{R}^{h \times |\vec{x}|}$ and $w_{oh} \in \mathbb{R}^{|L| \times h}$ and expects a feature vector as input. For the forward pass, matrix multiplications together with an activation function form a model function $m_{w_{hx},w_{oh}}(\vec{x}) \in \mathbb{R}^{|L|}$ that produces an output vector of probabilities:

$$sig(x) = (1 + e^{-x})^{-1},$$

$$m_{w_{hx},w_{oh}}(\vec{x}) = \underbrace{sig(w_{oh} \cdot \overbrace{sig(w_{hx} \cdot \vec{x})}^{a_{hx}})}_{a_{oh}}.$$

For preparation, we create the necessary tables for weights as well as the input matrix in SQL and define the sigmoid as an SQL function (see Listing 8.27). Afterwards, the forward pass is computed using an ArrayQL statement (see Listing 8.28).

```
CREATE TABLE input(i INT PRIMARY KEY, v FLOAT);
CREATE TABLE w_hx(i INT, j INT, v FLOAT, PRIMARY KEY (i,j));
CREATE TABLE w_oh(i INT, j INT, v FLOAT, PRIMARY KEY (i,j));
INSERT INTO ...
-- helper function
CREATE FUNCTION sig(i FLOAT) RETURNS FLOAT AS $$ SELECT 1.0/(1.0+exp(-i));$$ LANGUAGE 'sql';
```

LISTING 8.27: Preparation for the neural network in SQL-92.

```
SELECT [i],[j], sig(v) as v FROM w_oh * (SELECT [i],[j], sig(v) as v FROM w_hx * input);
```

LISTING 8.28: Forward pass in ArrayQL.

8.4.3 Logical Optimisations

Implemented within a relational database system, ArrayQL benefits from query optimisation by design. The operators undergo logical optimisations, inherited from the nature of the relational operators, including cost-based query plan reordering based on heuristics on the table sizes. This does not include mathematical optimisations that make use of, e.g., the symmetry property of matrices. We discuss the logical optimisations theoretically and show query plan reordering using multiplication of three matrices as an example.

Optimisation Steps

Database systems optimise queries logically by breaking up conjunctive predicates and pushing them down together with projections and changing the join order. We outline these optimisation steps with regard to ArrayQL operators.

- *Conjunctive predicate break-up and predicate push-down:* affects the filter and the rebox operator, as both are translated into selections. Filtering is similar to a selection. The rebox operator allows us to ignore all tuples outside the specified range.
- *Projection push-down*: concerns the apply and shift operator, that are both translated into projections. This is only beneficial for unbound attributes or when the query optimiser covers mathematical transformations.
- *Join ordering:* applicable to the combine operator and the inner dimension/extended join that are translated into joins.
- *Other:* Beside projections, also aggregations should consider mathematical transformations and be pushed down if possible, as required by reduce. The rename operator is only relevant for the data flow.

Cost-Based Query Plan Reordering

We demonstrate cost-based query plan reordering by an example of a three-way matrix multiplication. Given three matrices $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times o}, C \in \mathbb{R}^{o \times p}$, associativity allows computing their product $ABC \in \mathbb{R}^{m \times p}$ either as $(AB)C$ with $AB \in \mathbb{R}^{m \times o}$ or as $A(BC)$ with $BC \in \mathbb{R}^{n \times p}$. This results in two different operator plans (see Figure 8.5) with the two joins as common elements but with a different order concerning the aggregations. Although logical optimisation might push down the matrix subproduct out of projection and aggregation above the first join, the query optimiser must be aware of distributive properties. This allows the optimiser to transform the statement $\sum_{j'=1}^{j} a_{ij'} \sum_{k'=1}^{k} b_{j'k'} c_{k'l}$ over $\sum_{j'=1}^{j} \sum_{k'=1}^{k} a_{ij'} b_{j'k'} c_{k'l}$ to $\sum_{k'=1}^{k} c_{k'l} \sum_{j'=1}^{j} a_{ij'} b_{j'k'}$, when needed.

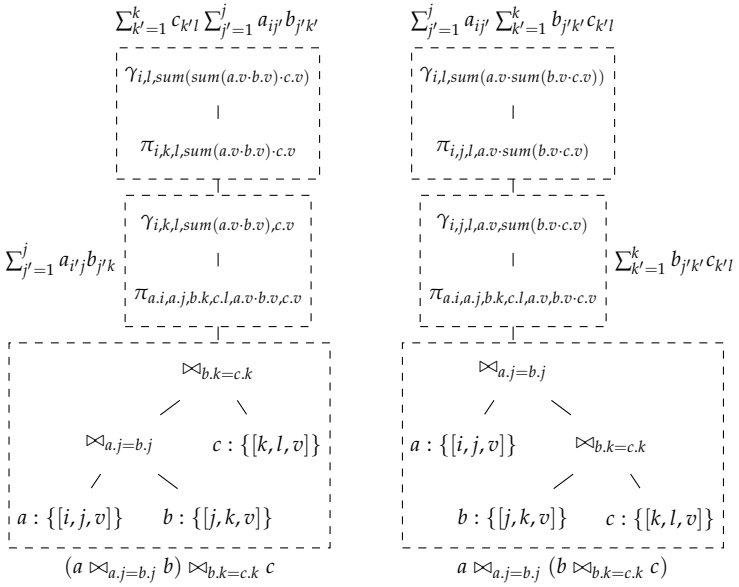


FIGURE 8.5: Unoptimised operator plan for three-way matrix multiplication of $(AB)C$ (left) and $A(BC)$ (right): the join on three relations might be reordered depending on the cardinalities. The projection and aggregation for the matrix subproduct can be pushed down above the first join.

HyPer and Umbra are using index-based heuristics for join order optimisation [83]. As index-based heuristics exploit index structures to calculate join selectivities, they estimate join cardinalities more precisely. This is ideally suited to a relational matrix representation

with an index on the dimensions used for joining. Given the densities $ds_a, ds_b, ds_{ab} \in [0,1]$, the selectivity of a join is given as $sel(A \bowtie B) = \frac{|A \bowtie B|}{|A||B|} = \frac{s_{ab} \cdot m \cdot o}{ds_a \cdot m \cdot n \cdot ds_b \cdot n \cdot o} = \frac{ds_{ab}}{n^2 \cdot ds_a \cdot ds_b}$ with ds_{ab} as the single unknown parameter.

8.5 Evaluation

For evaluation we measure the performance of ArrayQL under geo-temporal and linear algebra workloads. For the first use-case, we consider linear algebra extensions for database systems. For the latter, we compare the performance of ArrayQL in Umbra to those of popular array database systems. This section first discusses the performance of matrix operations before we proceed with the ArrayQL algebra on geo-temporal datasets.

System: All measurements have been conducted on a machine running Ubuntu 20.04 LTS, equipped with six Intel Core i7-3930K CPUs running at 3.20 GHz, and offering 64 GB of main-memory.

Data: We benchmark with the New York taxi dataset[1], the science benchmark SS-DB [33] and randomly generated data.

Competitors: The chosen competitors within popular array database systems are RasDaMan (version 10.0.0), MonetDB SciQL and SciDB (version 19.11) to benchmark geo-temporal applications. To benchmark linear algebra, we pick RMA as MonetDB's extension for linear algebra and MADlib (1.17.0 release) as an extension on top of PostgreSQL version 12.2.

8.5.1 Linear Algebra

This section discusses the performance of basic matrix operations as well as compound operations, either expressed in ArrayQL within Umbra or in SQL within its competitors, MADlib and RMA. MADlib provides two different representations as it distinguishes between arrays and matrices: for arrays, linear algebra operations are applied to the PostgreSQL array type, for matrices, operations expect a table in relational representation. Thus, MADlib's sparse relational representation corresponds to the underlying one for ArrayQL. In contrast, RMA uses a tabular representation. This section justifies the applicability of a relational representation for linear algebra without losing performance.

Matrix Algebra

This subsection presents the performance of micro-benchmarks (matrix addition and gram matrix computation) for the three matrix types (MADlib arrays, MADlib sparse matrices, RMA) against ArrayQL.

RMA's tabular representation depends on the database schema (the first dimension corresponds to the attributes, the second to the number of tuples). For benchmarking purposes, RMA provides a Python script, that creates the schema, inserts as many tuples as the specified size for the second dimension and creates SQL statements for matrix addition and gram matrix computation. For comparison, we add support to create statements for MADlib and ArrayQL and fill the relations with the same data.

We disable autocommit to measure execution time only, as it would slow down RMA dramatically. In the following, we measure the impact of the size and sparsity of a matrix on the runtime when performing matrix addition and gram matrix multiplication.

Figure 8.6 shows the runtime needed for matrix addition ($X + X$), when varying the sparsity, and on dense arrays, when varying the input size. With increasing size, ArrayQL computes the matrix sum faster than RMA. RMA's compute time consists of optimisation and runtime, both increase with the size of a matrix. When varying the sparsity, MADlib matrices and Umbra benefit from sparse matrices, since zero values simply do not exist. RMA needs constant runtime with increasing sparsity as sparse and dense matrices consume the same space in a tabular representation.

Matrix addition on MADlib matrices performs the worst, whereas the same operation on MADlib arrays performs the best. This is reasonable, as the aggregation time needed to create arrays out of its relational form is not considered.

[1] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

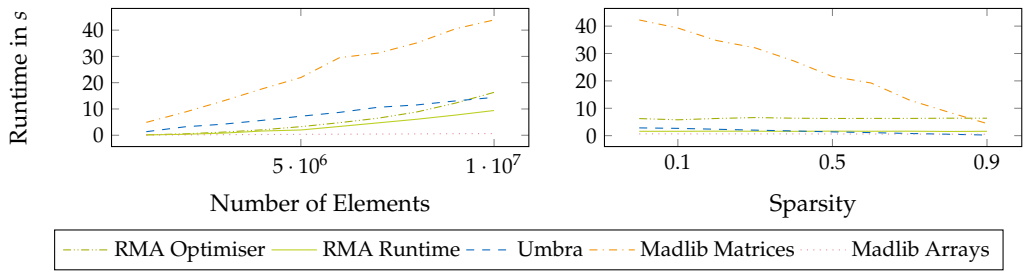


FIGURE 8.6: Evaluation of matrix addition: varying the number of elements in a dense array or the sparsity of an array with 10^6 elements.

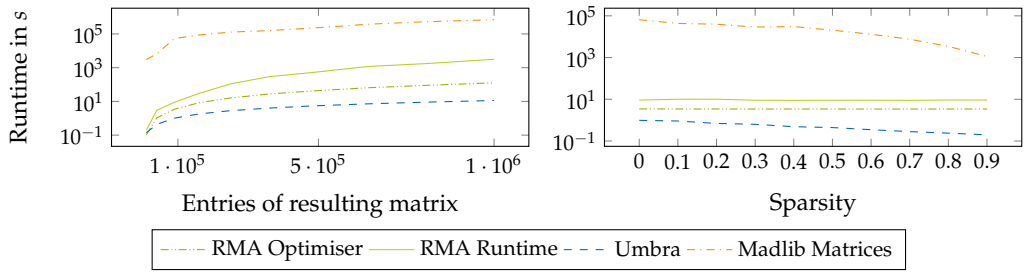


FIGURE 8.7: Evaluation of gram matrix computation: varying the number of elements in a dense array and the sparsity of a resulting matrix with 90000 entries.

Gram matrix computation ($X \cdot X^T$, see Figure 8.7) yields similar results: the higher the sparsity, the lower the runtime when handling MADlib matrices as well as within ArrayQL in Umbra. MADlib does not allow to transpose arrays, so gram matrix computation is not possible. Again, RMA needs constant compute time and, as the transposition is more expensive in a tabular representation, it is slower than Umbra.

When varying the input size, multiplication on MADlib matrices takes the most time. Multiplication in ArrayQL results in the shortest execution time as it is based on Umbra's relational algebra.

In summary, ArrayQL in Umbra benefits from sparse matrices as well as the performance of an in-memory database system. Therefore, our relational representation shows comparable performance to existing database extensions for linear algebra.

Linear Regression

We solved linear regression to benchmark composed matrix operations on a synthetic dataset. This section compares Umbra's performance with ArrayQL when using linear algebra only to the one of MADlib, which provides a dedicated table function to compute the optimal weights. Figure 8.8 shows the runtime of both systems when either varying the number of input tuples or the number of attributes. Our solution for ArrayQL—relational algebra with a table function for matrix inversion—outperforms MADlib's table function for linear regression only for a small number of input tuples or attributes. To further investigate the performance in Umbra, Figure 8.9 breaks down the runtime into the individual sub-operations. With increasing number of input tuples, the impact of the materialising inverse function on the runtime decreases as the inverted matrix $(X'^T X')^{-1}$ becomes relatively small. Most time is spent on the aggregation part of each matrix product (summation). Instead of using matrix algebra, a dedicated equation solve function can compute linear regression more efficiently. But calling an optimised equation solve function has the downside of materialising the input first. Nevertheless, this experiment has shown the ability to solve numerical problems when suitable table functions are available. The conception of a non-materialising table function for the matrix inversion [140] and a non-materialising equation solve function is for future work. The hash table used for summation can be further optimised: when the number of keys (indices) is known beforehand, the memory can be preallocated and the tuples prepartitioned for efficient access.

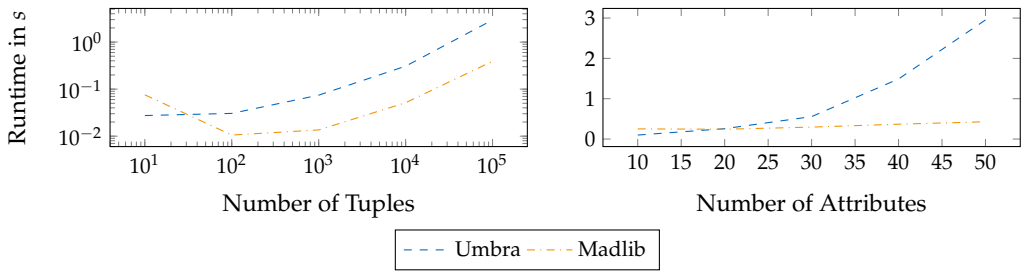


FIGURE 8.8: Runtime for solving linear regression when varying the number of tuples (50 attributes) or when varying the number of attributes (10^5 input tuples).

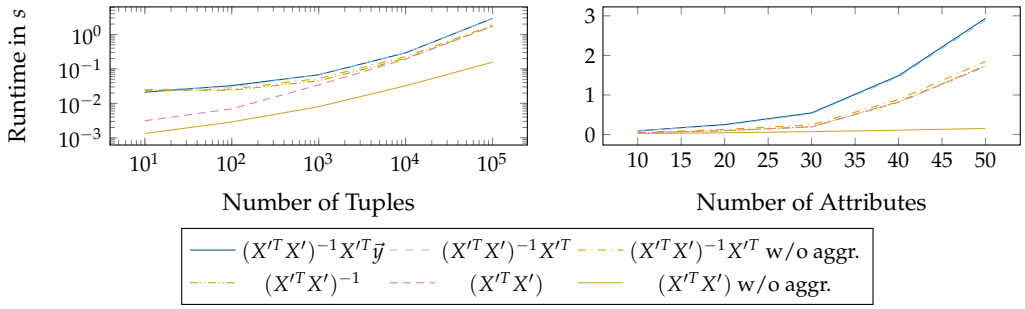


FIGURE 8.9: Runtime within Umbra broken down by operation for solving linear regression when varying the number of tuples (50 attributes) or when varying the number of attributes (10^5 input tuples).

8.5.2 Array Operations

This section compares the performance of ArrayQL in Umbra to the one of popular array database systems. RasDaMan[2] and MonetDB SciQL-2-NetCDF[3] ran natively on the system, a Docker container running Ubuntu 16.04 was used for SciDB[4]. Goetz [51, 52] tested basic operations on the New York Taxi dataset, array operations with the SS-DB benchmark and with synthetic data, which are referenced for completeness.

New York Taxi Data

Using the New York taxi data[5] (624 MB), we benchmarked real-world queries (see Table 8.4) on one-, two- and ten-dimensional arrays.

Queries Q1, Q3 and Q7 benchmark projections, whereas queries Q2, Q4, Q5, Q6 and Q8 benchmark aggregation functions like summation, average and count, and queries Q9/Q10 modify the array bounds. In detail, Q1 requests one attribute (`vendorID`). Q2 sums up the total distance driven. Q3 computes the distance ratio of one ride to the total distance driven. Q4 returns the maximum duration of a trip. Q5 returns the average total amount (payment), whereas Q6 calculates the average payment per customer excluding trips with no passengers. Q7 returns all attributes of trips with four or more customers. Q8 retrieves the frequency of a specific payment method. Q9 reboxes the array indices (slice and shift). Q10 slices the array to return a subarray.

Figure 8.10 shows the runtime of the presented queries on the New York taxi dataset stored as a one- or two-dimensional grid. To be comparable to the array database systems, which store the data as a dense grid, we added a synthetic key to the data in Umbra. ArrayQL within Umbra performs well on computing aggregates (Q2, Q4, Q5, Q6 and Q8) and slicing the array (Q10). The runtime includes the time for printing to `/dev/null`, which influenced the runtime for queries returning multiple array entries (Q1, Q3, Q7, Q9) as the index was attached to every value. The compilation time of ArrayQL queries was negligible

[2] https://doc.rasdaman.org/index.html
[3] https://dev.monetdb.org/hg/MonetDB/shortlog/SciQL-2-NetCDF
[4] https://github.com/rvernica/docker-library
[5] https://nyc-tlc.s3.amazonaws.com/trip+data/yellow_tripdata_2019-12.csv

TABLE 8.4: ArrayQL queries on the New York taxi dataset [51].

Q1	`SELECT VendorID FROM taxiData;`
Q2	`SELECT SUM(trip_distance) FROM taxiData;`
Q3	`SELECT 100.0*trip_distance/tmp.total_distance FROM taxiData, (SELECT SUM(trip_distance) as total_distance FROM taxiData) as tmp;`
Q4	`SELECT MAX((tpep_dropoff_datetime - tpep_pickup_datetime) + (end_time - start_time)) FROM taxiData;`
Q5	`SELECT AVG(total_amount) FROM taxiData;`
Q6	`SELECT AVG(total_amount/passenger_count) FROM taxiData WHERE passenger_count <> 0;`
Q7	`SELECT * FROM taxiData WHERE passenger_count>=4;`
Q8	`SELECT COUNT(*) FROM taxiData WHERE payment_type=1;`
Q9	`SELECT [0:1048574] as i, * FROM taxiData[i+1];`
Q10	`SELECT [42:42000] as i, * FROM taxiData[i];`

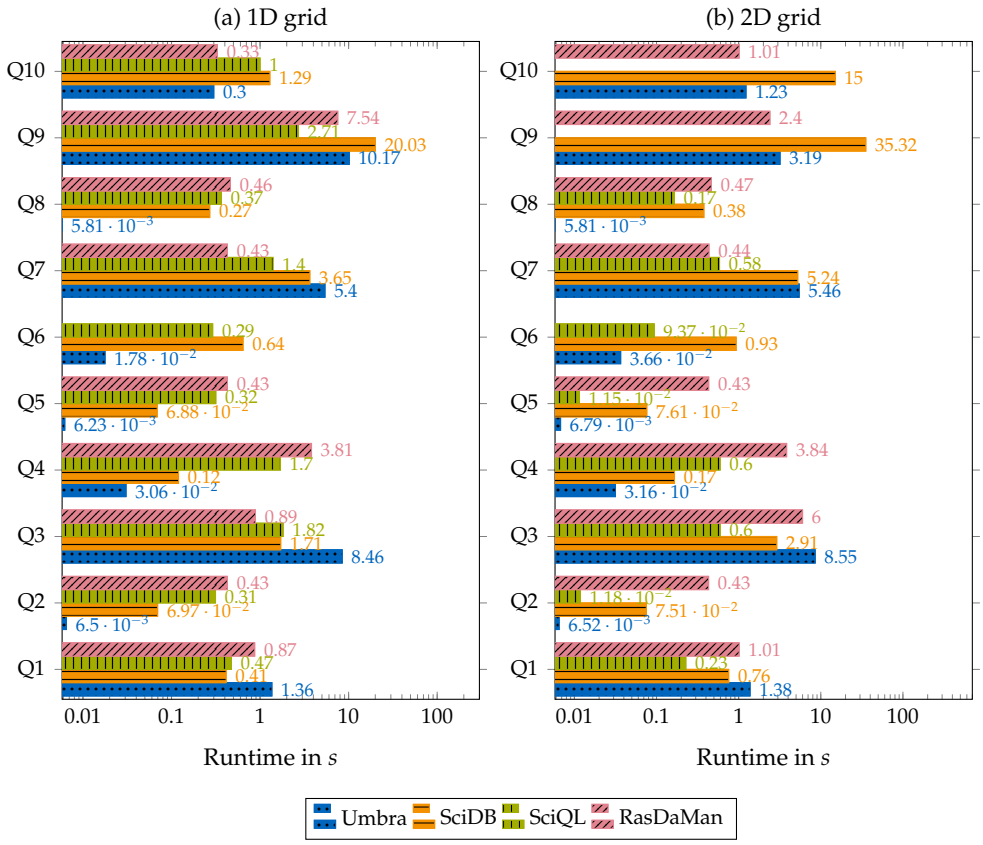


FIGURE 8.10: New York taxi dataset with (a) one- and (b) two-dimensional indices [51].

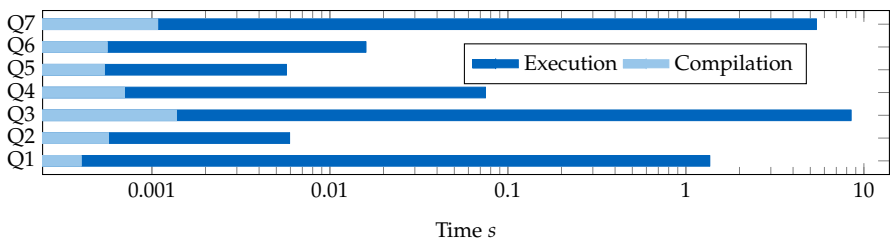


FIGURE 8.11: Compilation time vs. runtime of selected ArrayQL queries in Umbra [51].

(see Figure 8.11). In comparison, RasDaMan performed well when changing the dimensions and retrieving data without further computations (Q7). SciDB performed better than RasDaMan (Q1, Q2, Q4, Q5) as long as the query did not alter the dimensions, as its *reshape* slows down the runtime (Q9, Q10). In summary our ArrayQL extension performed well on aggregations and on accessing an subarray due to an index on the dimensions.

TABLE 8.5: Multi-dimensional queries (New York taxi data) [51].

SpeedDev	`SELECT MAX(3600.0*(tmp3.v3-tmp1.v)) as speed FROM (SELECT pickup_day, AVG(trip_distance/total_duration) as v FROM taxiData WHERE total_duration<>0 GROUP BY pickup_day) as tmp1, (SELECT AVG(trip_distance/total_duration) as v3 FROM taxiData WHERE total_duration <> 0) as tmp3;`
MultiShift	`SELECT [pickup_day] as a, ..., * FROM taxiData[a+1, ...]`

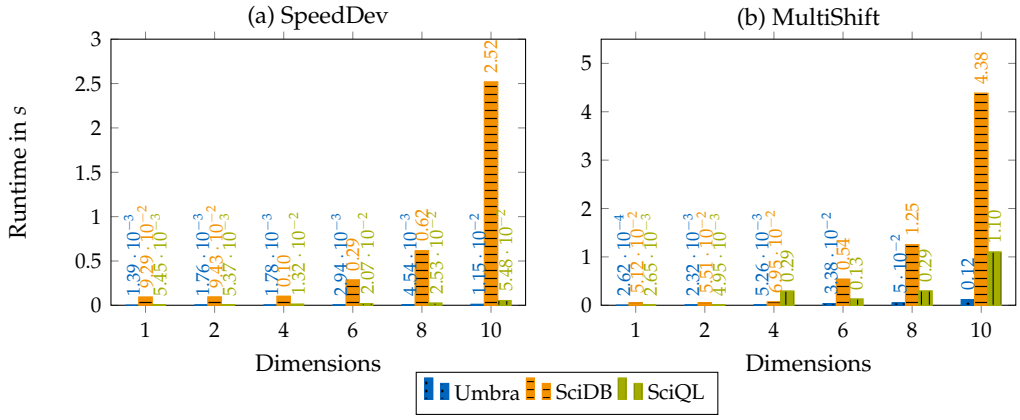


FIGURE 8.12: Impact of dimensionality on the runtime [51].

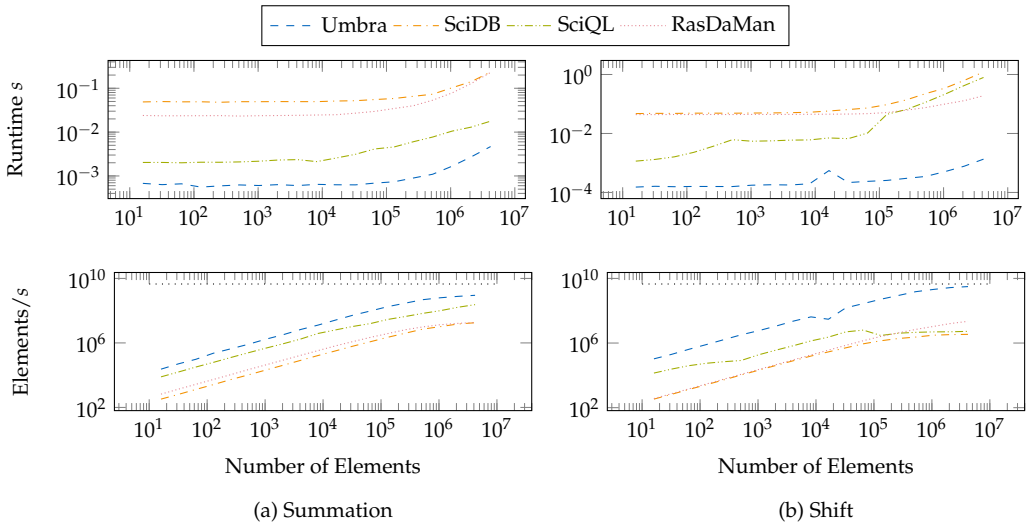
To measure the impact of involved dimensions on aggregations (see Table 8.5), we also stored the taxi data as a ten-dimensional array. Query *SpeedDev* calculates "the maximum deviation of the average speed per day" [51] compared to the average speed of the whole dataset. Query *MultiShift* shifts all array's dimensions. Figure 8.12 shows the runtime in dependency on the number of dimensions. For both queries, the runtime on all tested systems scaled linearly with the number of dimensions. Umbra and MonetDB showed similar performance for the query *SpeedDev*, both outperforming SciDB.

The query *MultiShift* showed that also MonetDB SciQL performed well on multi-dimensional arrays. Umbra outperforms both competitors in compute time (the time for printing the indices not considered). SciDB was slower independent of the number of dimensions.

Random Data

Figure 8.13 shows the runtime and throughput on two-dimensional arrays with synthetic data and an increasing number of elements. For both tests, summation and shifting the indices, the runtime scales linearly with the number of elements. Umbra is the fastest system when performing aggregates. The upper constant lines in the lower diagrams display the maximum throughput of $4.5 \cdot 10^9$ elements per second (based on a measured[6] memory bandwidth of 36 GB/s divided through 8 B, the size of a double precision floating-point number). This shows that ArrayQL in Umbra best approaches the maximum possible performance, with only a factor of ten in between.

[6] https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html



(a) Summation (b) Shift

FIGURE 8.13: Runtime and Throughput of an aggregation (summation) or shifting the indices of a two-dimensional matrix depending on the number of elements [51].

SS-DB Data

SS-DB is a benchmark for scientific workloads that simulates astronomical data for array-oriented processing. A data generator[7] synthesises three-dimensional data—one dimension identifies the tile, two dimensions determine a cell—with eleven attributes each.

The queries SSDBQ1/2/3 (see Table 8.6), adapted from a paper that compares SciQL and SciDB [103], compute the average of one attribute for 20 tiles. SSDBQ2 and SSDBQ3 do the same but consider fewer cells (50 % and 25 % of the original size). All queries were tested on an image of size tiny (58 MB), small (844 MB) and normal (3.4 GB).

Figure 8.14 presents the measured runtimes. For the first query, Umbra outperformed the competitor systems as it benefitted from an index on the dimensions. The other two queries combined aggregations and predicates with shifting the indices, which could be processed the fastest by Umbra and SciQL (the latter also performs well on aggregations). Adjusting the indices is expensive within SciDB, which slowed down its performance.

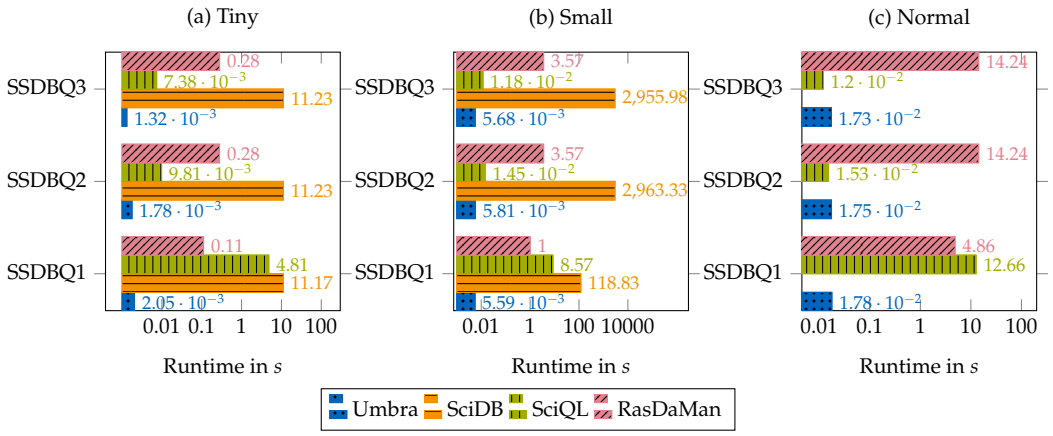


FIGURE 8.14: SS-DB: one image of size (a) tiny, 58 MB, (b) small, 844 MB, and (c) normal, 3.4 GB [51].

[7] https://www.xldb.org/science-benchmark/

TABLE 8.6: Performed queries on the SS-DB dataset [51].

SSDBQ1	ArrayQL:	`SELECT AVG(a) FROM ssDB[0:19]`
	AQL:	`SELECT AVG(a) FROM subarray(very_small,0,0,0,19,1599,1599)`
	SciQL:	`SELECT AVG(a) FROM ssDB[0:19]`
	RasQL:	`SELECT AVG_CELLS(r.a * (r.z<=19)) FROM ssDB as r`
SSDBQ2	ArrayQL:	`SELECT AVG(a) FROM ( SELECT [z], [x] as s, [y] as t, * FROM ssDB[0:19, s+4, t+4] WHERE s%2 = 0 AND t%2 = 0) as tmp GROUP BY z`
	AQL:	`SELECT AVG(a) FROM reshape( subarray(ssDB, 0,0,0,19,999,999), <a:int32, b:int32, c:int32, d:int32, e:int32, f:int32,g:int32,h:int32,i:int32,j:int32, k:int32> [z=0:19,1000000,0,x=4:1003,1000000,0,y=4:1003,1000000,0]) WHERE x%2=0 and y%2=0 GROUP BY z`
	SciQL:	`SELECT AVG(a) FROM ( SELECT [z], [x+4] as s, [y+4] as t, * FROM ssDB[0:19] WHERE x%2 = 0 AND y%2 = 0) as tmp GROUP BY z`
	RasQL:	`SELECT ADD_CELLS(shift(r.a * (r.z<=19 and mod(r.x, 2) = 0 and mod(r.y, 2) = 0), [4,4])) / COUNT_CELLS(shift((r.z<=19 and mod(r.x,2) = 0 and mod(r.y,2) = 0) , [4,4])) FROM ssDB as r`
SSDBQ3	ArrayQL:	`SELECT AVG(a) FROM ( SELECT [z], [x] as s, [y] as t, * FROM ssDB[0:19, s+4, t+4] WHERE s%4 = 0 AND t%4 = 0) as tmp GROUP BY z`
	AQL:	`SELECT AVG(a) FROM reshape( subarray(ssDB, 0,0,0,19,999,999), <a:int32, b:int32, c:int32, d:int32, e:int32, f:int32,g:int32,h:int32, i:int32,j:int32, k:int32> [z=0:19,1000000,0 ,x=4:1003,1000000,0, y=4:1003,1000000,0]) WHERE x%4=0 and y%4=0 GROUP BY z`
	SciQL:	`SELECT AVG(a) FROM ( SELECT [z], [x+4] as s, [y+4] as t, * FROM ssDB[0:19] WHERE x%4 = 0 AND y%4 = 0) as tmp GROUP BY z`
	RasQL:	`SELECT ADD_CELLS(shift(r.a * (r.z<=19 and mod(r.x, 4) = 0 and mod(r.y, 4) = 0), [4,4])) / COUNT_CELLS(shift((r.z<=19 and mod(r.x,4) = 0 and mod(r.y,4) = 0) , [4,4])) FROM ssDB as r`

8.6 Conclusion

In this chapter, we have integrated ArrayQL into a code-generating database system as another query interface and addressable inside SQL as user-defined functions. As this standardised array query language has not yet been integrated into a productive system, we completed its grammar specification and extended Umbra's query engine to accept ArrayQL statements. For that reason, we defined a relational array model and translated ArrayQL operators to relational algebra. We demonstrated the suitability of an array query language for geo-temporal data by the SS-DB benchmark and by the New York taxi data as a real-world example. Moreover this language can be used for linear algebra to compute machine learning tasks. For basic matrix operations, ArrayQL statements performed better than state-of-the-art linear algebra extensions for database systems, whereas materialising table-functions as needed for inversion slowed down the runtime. For geo-temporal tasks, ArrayQL outperformed traditional array database systems, when performing aggregations or filtering data by a predicate.

Chapter 9

Versioning in Main-Memory Database Systems

Parts of this chapter have been published in [67, 119, 123].

On the one hand, software engineering relies on version control systems such as CVS, SCCS [106], RCS [135], SVN and Git to allow distributed code development and to document the project's progress in commits and version tags. On the other hand, database systems are essential for efficiently handling huge amounts of data in index structures such as B*-[8] or Adaptive-Radix-Trees [81]. To unify both use cases, approaches such as temporal databases or dataset versioning aim to bind the validity of tuples to time instances or to allow collaborative dataset editing.

Online encyclopaedias such as Wikipedia rely on database systems to store the article's content but require dataset versioning to track the article's history. For this reason, MediaWiki, the software behind Wikipedia, implements its own version control to restore older versions on demand: one database table manages the pages' meta-information, another one contains their content with one entry per version and page. Besides the increased complexity when querying the data, this approach wastes space by storing redundant information as each revision changes an article only marginally.

So far, research has focused on versioning single datasets. But to fulfil the requirements for versioning a wiki system, a version control for database systems should be able to include multiple relations per single commit and respect referential integrity. When versioning one namespace of the Wikipedia encyclopedia, we expect a compression rate of up to 70 % when storing the differences only. Our estimation is based on the *Simple English* Wikipedia edition for which we computed the file and the edit differences for all page revisions (see Table 9.1).

TABLE 9.1: Estimation of saved storage when using compression techniques based on the *Simple English* Wikipedia page edit history dump of October 1, 2018.

	Size	Compression
Full Page Edit History	35.0 GiB	-
Current Version Only	1.1 GiB	-
History as File Diffs	14.0 GiB	59.77 %
History as Edit Diffs	9.4 GiB	72.71 %

Another use-case for dataset versioning arises from the reproducibility needed to validate scientific experiments: To reproduce the same results, the dataset used should remain unchanged. However, a production database system must allow updates and insertions. Therefore, for reproduction, any published experiment must provide the corresponding data as a separate snapshot when the underlying data store does not support dataset versioning.

In summary, a dataset versioning tool should serve the purpose of a simple data storage system with version-based access as well as the needs for transactional data processing on multiple relations. So it has to fulfil reference key constraints across multiple tables within each version.

Applying versioning and compression techniques to database systems eliminates the need for purpose-specific solutions, reduces the storage needed and allows comparable retrieval times (see Figure 9.1). But a complete integration into a database system includes the

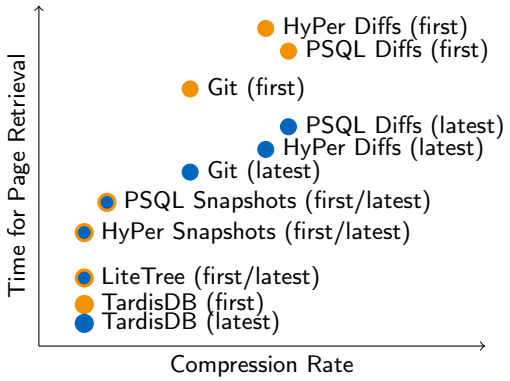


FIGURE 9.1: Sketch of the trade-off between storage savings (compression rate) and retrieval time: storing only one version snapshot and computing the others out of the changed differences (diffs) will reduce the amount of storage needed but will increase the retrieval time.

extension of SQL. An extension of this declarative language would not only increase the acceptance of database systems for dataset versioning but also facilitate software engineering projects: When the data layer takes care of dataset versioning, other layers are not concerned in the actual implementation.

We claim that database systems with full incorporated version control commands fill the gap caused by distributed data and knowledge management, and facilitate the versioning of wiki entries.

Starting at extending single tables—known from OrpheusDB [60]—to multiple relations per version, we present *MusaeusDB* as a stand-alone tool along the database system. It implicitly sends SQL commands, manages the versions in meta tables and benefits from the database systems' user rights management, as tables are checked out in the user's namespace.

We make use of the gathered knowledge for *TardisDB*, an in-memory database system with incorporated SQL commands for database versioning that deeply integrates versioning based on additional bitmaps for each tuple. It is an open-source project[1] for a code-generating in-memory database system that produces LLVM code according to the producer-consumer model [99]. Its table scan operator has been modified to produce only tuples for a selected branch. Therefore, each branch maintains a bitmap for every table, which denotes each tuple's visibility. Also, multi-version concurrency control is used to track each tuple's history. TardisDB is the first project that combines multi-version concurrency control with bitmaps for branching.

In our *TardisBenchmark*, we evaluate *TardisDB* and the different storage approaches developed for database systems, and compare their runtime to those of the version control system *Git*.

The main contributions of this chapter are:

- Extending version control on top of database systems to include multiple tables instead of single datasets per version.
- The architectural blueprint for integrating versioning inside a main-memory database system.
- The SQL extension for versioning, including an additional SQL statement for branch creation and an optional keyword as part of the from-clause to determine the branch.
- Named branches that can be created using existing SQL keywords and fit seamlessly into declarative statements.
- A versioning data type that compresses multiple versions within a single attribute.
- A benchmark, which covers dependencies between relations and evaluates different storage techniques in respect for the querying performance.

[1] https://github.com/tum-db/TardisDB

The chapter is structured as follows: Section 9.1 introduces *MusaeusDB*, our SQL-based prototype, and its architecture comprising the schema, syntax for the user input and storage conventions. Section 9.2 proceeds with the architecture of *TardisDB*, for which we adapt the table scan operator and the multi-version concurrency control of a main-memory database system to allow branches and to store multiple states of the database. In Section 9.3, we propose an extended SQL grammar, that is downward compatible with SQL-92, but allows branch creation and retrieving tables by version. Afterwards in Section 9.4, we explain a demonstration setup, with an interactive web interface that allows users to create branches, examine lineages, query and modify the data. To benchmark the prototypes, we will explain how we extract Wikipedia's page edit history to create a version control benchmark in Section 9.5. The benchmark introduces delta compression, which we integrate into the Umbra database system in Section 9.6. The evaluation section uses the benchmark for measuring time performance impacts of different versioning techniques on classical relational as well as modern main-memory database systems compared to storage savings (Section 9.7). Section 9.8 concludes by the performance and the achieved compression ratios and proposes to combine the different approaches.

9.1 MusaeusDB: Versioning using SQL

The SQL prototype—called *MusaeusDB* [74]—rely on the extended schema of *OrpheusDB* to combine changes over multiple relations as one version. It was originally developed by Kleiner [74] for PostgreSQL, then extended for use with HyPer and benchmarked as part of this thesis. The key idea is to keep the initial data tables unchanged but with an added record identifier (*rid*) and to store the information about versions in separate tables. As multiple tuples across different versions may share the same primary key, the *rid* is needed as the new primary key for all tuples.

The version table manages the corresponding *rid*s for each version and each table (we adopt the *rlist* as array-like structure but the table can be easily normalised using `unnest`). For each commit, the meta table contains information such as the commit message or the parent version. In comparison to *OrpheusDB*, we extended the version table by one attribute referencing the table in question. Instead of only one tuple, it contains as many tuples per version as tables involved. During query processing, we need an additional join predicate to retrieve the tuples of a certain version. Figure 9.2 shows the tables responsible for managing

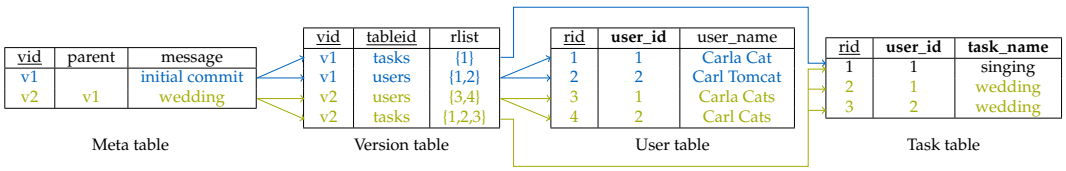


FIGURE 9.2: Schema: Version table and meta table for managing the commits on the left; tables containing the data on the right; the record id serves as a key for every tuple.

the commits and the tables containing the data. In this example, the meta table hosts two commits, one initial and one descending; the tuples concerned are stored in the version table. For example, the name of the users has changed after they got married, so the user table's attribute *name* has been updated and "*wedding*" was added to the task table. The data tables now have a unique artificial key (*rid*) as the original keys will not suffice as primary keys, but are restored on a table checkout. Indexing *rid* as primary key allows fast join processing even for a huge number of entries.

MusaeusDB stores datasets in *public repositories* and allows users to clone them. It benefits from the conception of database systems as they provide multiple databases per server (at least one for each user) with named schemas per database. In fact, a *public repository* is a separate namespace (a named schema) of a public database. Each user is allowed to check out a certain version in a private namespace of his/her database and may modify the tuples locally. Afterwards, the changes get propagated by a commit to the origin repository.

In Figure 9.3, we see the distinction between public and user databases with separate namespaces for the repositories. Each namespace represents one repository. The default

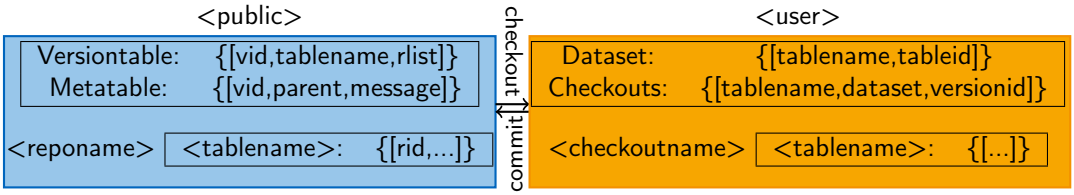


FIGURE 9.3: Distinction between global and local (user) space in *MusaeusDB*: The global space maintains a separate namespace for each repository, relations can be checked out for modifications in the user's namespace.

namespace of the public database hosts all versions and their meta information. On a *checkout*, all tuples belonging to the dedicated version are checked out into tables that are created for that purpose in the specified namespace. To keep track of all tables currently checked out, all checkouts are documented in the default namespace of the user's database. *MusaeusDB* is based on database systems implementing the PostgreSQL interface and works as a separate tool. For commits and checkouts it uses the pqxx library[2] to connect to the database server. The source code has been made publicly available[3]. In the following sections, we will introduce the SQL commands behind init, checkout and commit.

9.1.1 init

The *init* command prepares the tables of an existing namespace for versioning comparable to `git init`. The command expects the name of the destined global repository and the name of a schema with its tables to be prepared for versioning and collaborative working (see Listing 9.1). *MusaeusDB* generates the SQL commands to endue each tuple with an *rid* as its new primary key and to create a global table in the designated global namespace to which each of the source's relations can be copied.

```
1 $ ./musaeus init <public>.<reponame> <user>.<localreponame>
```

LISTING 9.1: MusaeusDB: init command: this takes the local database schema as source argument to copy it to a public database schema, which allows versioning.

9.1.2 checkout

The *checkout* command works contrary to the *init* one: this takes the name of the global namespace and copies the tables to newly created ones in a designated private namespace (see Listing 9.2). The global *rid* is hereby omitted and the original primary keys are restored.

```
1 $ ./musaeus checkout <public>.<reponame> <user>.<localreponame>
```

LISTING 9.2: MusaeusDB: checkout command: it takes the public database schema name as source argument to copy the tables to the schema given as second argument.

9.1.3 commit

The *commit* command updates the global repository with changed, inserted or deleted tuples. It takes the name of the source's namespace and a commit message (see Listing 9.3). It treats all changes as one whole commit and pushes the updates to the origin. The rlist for the new version is copied from the ancestor one except the rids of the changed or deleted tuples. For every changed or inserted tuple, a new rid is created and added to the rlist. As the command is translated into one atomic transaction, the tool ensures that commits are processed atomically, and respects referential integrity as local tables inherit the parent's database schema. This is useful as MediaWiki stores the page title separately from the content, but using references.

[2] http://pqxx.org/development/libpqxx/
[3] https://gitlab.db.in.tum.de/tardisDB/musaeusDB

```
$ ./musaeus commit <user>.<localreponame> <commitmessage>
```

LISTING 9.3: MusaeusDB: commit command: the changes made in the given schema name are updated in the remote repository.

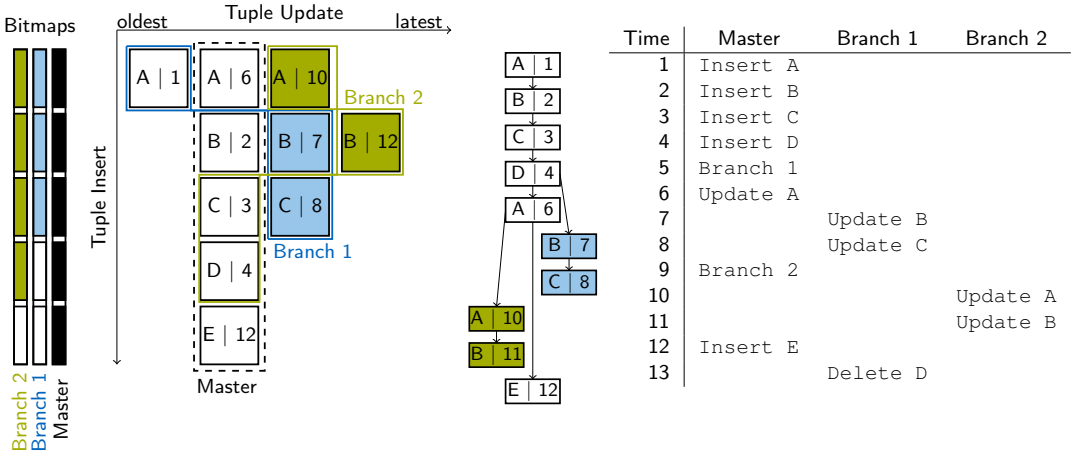


FIGURE 9.4: Left: Bitmaps for each branch (y-axis) mark contained tuples per table, whereas version chains track their history (x-axis), tuples of the master branch are stored in a column-oriented manner. Middle: Descendance tree. Right: Corresponding insert, update, delete and branching history.

9.2 Versioning in Code-Generating Database Systems

TardisDB follows the concept of a code-generating main-memory database system: It produces LLVM code according to the producer-consumer concept [99] where every operator demands the underlying ones recursively to generate code. Versioning now affects the table scan operator at the bottom of each operator tree: During compile time, code for checking the bitmap as well as retrieving the correct tuple out of a version chain has to be generated. For every table and every branch, we maintain a bitmap, where a bit indicates the tuple's affiliation to a branch. On a table scan, a tuple will be produced only if the corresponding bit is set. When a branch is created, the parent bitmap is simply copied for every table. On an insert statement, all bitmaps are enlarged but the new bit is set for the affected branch only. On a delete statement, the corresponding bit is reset.

The bitmap indicates included tuples only but to track different versions of a tuple, we maintain version chains as used for multi-version concurrency control. This is the first time that multi-version concurrency is combined with branching to not only track a tuple's modification history but to provide access to a set of tuples that form one version. This approach tends to densely populated bitmaps even on update-heavy workloads, which increases the total scan performance.

We define one prioritised branch, the master branch, whose tuples are stored in a regular column store and updated in-place. Each tuple is given two pointers to form a double-linked list of version entries and a timestamp, which is inherited from the branch that created the version.

Each branch, in turn, is given a timestamp at its creation, which is also used to retrieve the correct entry in the version chain: Formally, we define a predicate $active(t, b)$ for each entry t and branch b to indicate visible entries:

$$active(t,b) \Leftrightarrow created(b,t) \vee \bigvee_{p \in parent(b)} active(t,p) \wedge ts(t) < ts(b).$$

An entry is visible for a branch when either the branch itself ($created(b,t)$) or a parent branch ($parent(b)$) before furcation ($ts(t) < ts(b)$) has created the entry. The first active entry in the chain is the latest visible one and is returned.

Figure 9.4 visualises the history for five tuples on one master and two descending branches: Initially, four tuples were inserted before *Branch 1* is created with timestamp 5. The tuples for the master branch are stored column-wise (dashed line) and each contains a pointer to previous (left) and newer (right) versions. An update on the master branch changes the value in-place (`A|6`) and creates an entry for the previous version (`A|1`), an update on a descending branch just creates an entry on the right (`B|7`). When *Branch 1* requests tuple `A`, it receives `A|1` as `A|6` is only active within *Master* and *Branch 2*. Whereas two versions for tuple `B` are active within *Branch 1*, which is why it accesses the newer one (`B|7`). Instead of deleting a tuple, the corresponding bit is reset (`D` on *Branch 1*).

9.3 SQL Extension

We now extend SQL (see Listing 9.4) to address the extended table scan operator. This requires a statement to create branches as well as an extension to specify the version for each table that is part of select, update, insert or delete statements. By default, regular SQL-92 queries without any adaptations are applied to the *master* branch to ensure downward compatibility.

```
1 CREATE TABLE users (id INT PRIMARY KEY, name TEXT);
2 CREATE TABLE things (id INT PRIMARY KEY, name TEXT, user INT REFERENCES users(id));
3 INSERT INTO users VALUES (1, 'Alice');
4 INSERT INTO things VALUES (21, 'printer', 1);
```

LISTING 9.4: Example of tables created and filled with SQL.

To create a new branch, we add a statement to fork branches from existing ones (see Listing 9.5). This statement only expects the name of the parent branch and the created one. It does not require any further information as branches affect all tables to fulfil foreign key constraints. Creating a new branch is necessary to maintain access to a certain database state whenever this version should be preserved.

```
1 CREATE BRANCH mybranch FROM master;
```

LISTING 9.5: Statement to create the branch *mybranch* from the parent *master*.

To query tables on the created branch, we propose the keyword `VERSION` behind each table that is part of an insert, select, update or delete query (see Listing 9.6).

```
1 INSERT INTO users VERSION mybranch VALUES (2, 'Bob');
2 UPDATE things VERSION mybranch SET user=2 WHERE id=21;
3 SELECT * FROM users VERSION mybranch;
```

LISTING 9.6: Insert, update and select statements on tables of a certain branch using the `VERSION` keyword.

Although branches affect all tables, explicitly specifying the branch enables access to tables across different branches in one statement. This allows merging tables of different branches using ordinary SQL statements. For example, a full outer join allows conflicting tuples sharing the same primary key to be identified (see Listing 9.7).

```
1 SELECT a.id, COALESCE(a.name, b.name)
2 FROM users VERSION master as a FULL OUTER JOIN
3      users VERSION mybranch as b ON a.id=b.id
```

LISTING 9.7: Merging tables.

Finally, after changes in a branch have been merged or become outdated, we propose a delete statement to free the allocated resources (see Listing 9.8). We propose a garbage collection [19] to remove versions that are not contained within a branch anymore.

```
1 DELETE BRANCH mybranch;
```

LISTING 9.8: Branch deletion statement.

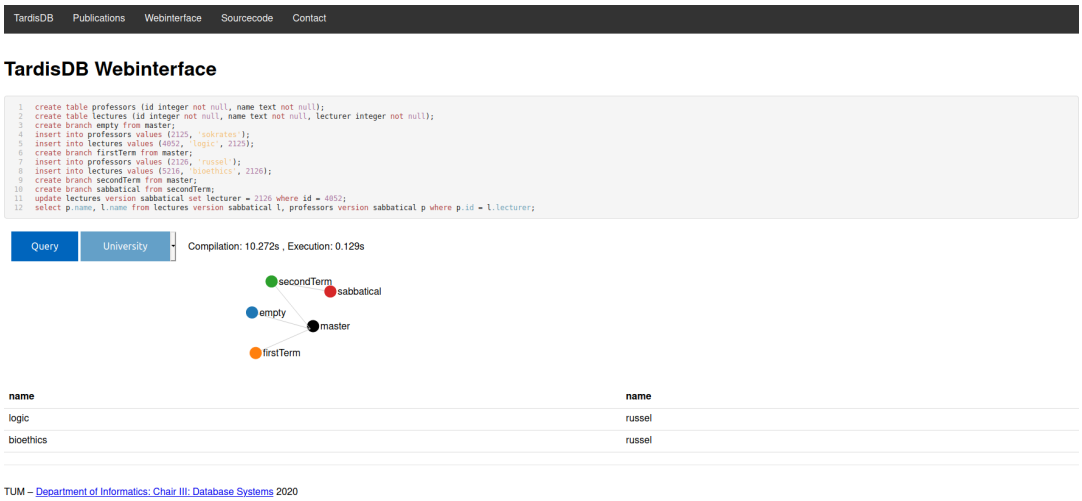


FIGURE 9.5: TardisDB web interface: An interface allows SQL queries including branch creation to be formulated. The chart in the middle displays the lineage of all available branches; the result table is shown at the bottom.

9.4 Demonstration Setup

We have created an interactive web interface[4] to demonstrate extended SQL on TardisDB (see Figure 9.5). Within the web interface, we allow users to create tables, insert data and create branches. The lineage of created branches is visualised graphically. Of course, the SQL interface allows querying the data including joins over different branches. The result together with the query time is displayed afterwards. For demonstration purposes, we started a TardisDB instance on a remote server that creates a new database instance for each client. This allows everyone to try out the extended SQL on their own device. We provide examples together with a selected CSV file as input data to demonstrate the performance of the available operators.

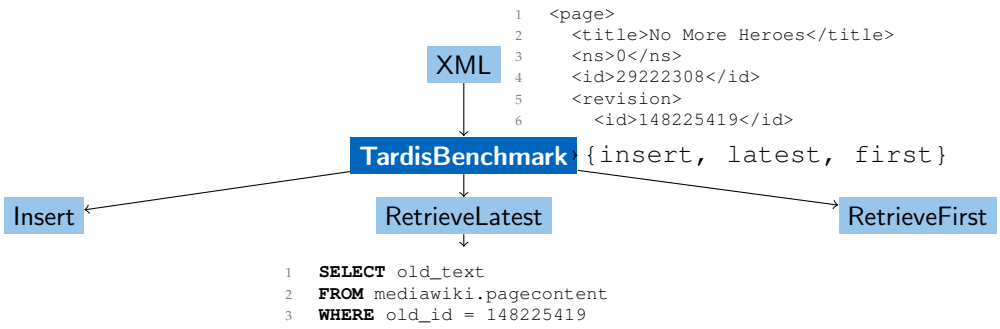


FIGURE 9.6: Conception of the *TardisBenchmark*: it allows running retrieval or insertion queries, the latter requires an XML file out of the page edit history.

9.5 TardisBenchmark

This section presents the *TardisBenchmark* based on the Wikipedia page edit history. We have chosen the MediaWiki schema as it is the most common example of a versioned database with reference key constraints, which we will introduce in the following. Afterwards, we will explain the different aspects and operations that we consider for our benchmark and

[4] http://tardis.db.in.tum.de

what storage approaches we tested with the operations. The TardisBenchmark has been implemented by Karnowski [65] and is published as open-source[5].

9.5.1 MediaWiki Schema and Wikipedia Data

Wikipedia is the 13th most important web site in the world with 1,414,644 sites linking in [4]. Being a collaborative online encyclopedia to which everyone is allowed to contribute, Wikipedia relies on version control mechanisms that allow previous versions to be restored if the free edit rights are misused. Together with the fact that data and the architecture are freely available, Wikipedia is the ideal candidate for deriving a benchmark out of it.

The core of MediaWiki, the software behind Wikipedia, is a database schema (see Listing 9.9) out of the three relations *page* (the meta data of the articles), *pagecontent* (the actual content) and *revision* (references to former versions of an article). A current article can be retrieved by finding the page by title first (see Listing 9.10), then a foreign key points to the current page content (see Listing 9.11). We feed our benchmarking program with the database dumps in XML format. The dumps may consist of all latest versions or also of the complete edit history. In this manner, the benchmark can have a flexible workload up to 13 TiB, the size of the full English Wikipedia (1 August, 2018).

```
1 CREATE TABLE page (page_id INT PRIMARY KEY, page_title TEXT, page_latest INT);
2 CREATE TABLE revision (rev_id INT PRIMARY KEY, rev_page INT REFERENCES page (page_id),
      rev_text_id INT, rev_parent_id INT, rev_timestamp TIMESTAMP);
3 CREATE TABLE pagecontent (old_id INT PRIMARY KEY, old_text TEXT);
```

LISTING 9.9: Simplified MediaWiki schema [65]: *page* (all meta information), *pagecontent* (actual page content) and *revision* (history of former versions).

```
1 SELECT page_id, page_title, page_latest FROM mediawiki.page WHERE page_title = $1$ LIMIT 1
```

LISTING 9.10: Retrieve page by given title (`$1$`) [113].

```
1 SELECT old_text FROM mediawiki.pagecontent WHERE old_id = $1$ LIMIT 1
```

LISTING 9.11: Retrieve page content by identifier `$1$` [113].

```
1 CREATE TABLE page (page_id INT PRIMARY KEY, page_title TEXT, page_latest INT, page_len INT,
      page_text TEXT);
2 CREATE TABLE revision (rev_id INT PRIMARY KEY, rev_page INT, rev_parent_id INT, rev_timestamp
      TIMESTAMP, rev_text_id INT, rev_order INT);
3 CREATE TABLE pagecontent (old_id INT PRIMARY KEY, old_text TEXT);
```

LISTING 9.12: Modified schema (*Diff*) [65]: *page* contains the current content, *revision* manages the differences stored in *pagecontent*.

9.5.2 Operations

Our benchmark covers the most common use cases, which are the insertion of articles or retrieval of the latest or first version of an article. Figure 9.6 shows the conception of the benchmark with the XML file as input and the three different queries. The motivation for picking this set of benchmark operations is as follows [65]:

- **Insertion (*insert*).** When a page is initially created or updated, the page content will be inserted as a new tuple to the database system. So inserting articles is the fundamental operation on which any project relies on. We take the Wikipedia dumps with full page edit history as real world examples.
- **Retrieving the latest version (*retLatest*).** The common case an online encyclopedia is used or collaborative work is performed is the retrieval of the latest version of an article or the software. When using Wikipedia, users are always redirected to the current version. Former versions can be viewed on demand afterwards. So retrieving the latest version will form the majority of the work load.

[5] https://gitlab.db.in.tum.de/tardisDB/TardisBenchmark

- **Retrieving the first version (*retFirst*).** Retrieving the first (oldest) version of an article covers the retrieval of any prior version. This is important when storing former versions as differences to the latest article, as all deltas between the versions have to be applied. As another benefit, when we support the operations for retrieving both the first and the latest version of an article, we obtain the runtime of backward and forward delta-based versioning techniques, but we need to compute the differences in one direction only.

9.5.3 Storage and Compression

Within database systems, we consider storing the data as a snapshot or as differences, consisting of the changes between two versions only. In addition, we compare storing Wikipedia articles within Git and LiteTree.

The snapshot-based approach forms the classical way of storing every article as a new tuple in the manner of MediaWiki. We therefore rebuild the database schema out of *pagecontent*, *revision* and *page* to benchmark the performance. The operations are listed in further detail here:

- **insert.** The new content is always (on update or insert) added to *pagecontent*. When the page is initially created, a new tuple is added in *revision* and *page*. Otherwise on a page edit, we just update the references.
- **retLatest.** Retrieving the latest version means a join on *page* (stores the latest `pageid`) and *pagecontent*.
- **retFirst.** We retrieve the first version of an article by selecting the minimal timestamp of a revision in the table of that name.

The difference-based versioning reduces the amount of wasted storage by storing the latest version as a whole and the previous ones as text differences (see Listing 9.12). We adapt the MediaWiki schema to make it suitable for storing text differences, as we save the latest page content as an additional attribute of the *page* table and store the text differences in *pagecontent*. As the latest version is stored as a whole in *page*, this ensures that the page demanded most frequently can be retrieved quickly, with no need for joins. For an arbitrary version, we apply all differences starting from the latest, also called *backward deltas*.

- **insert.** On a page edit, the content in *page* will be updated and the difference from the previous one will be inserted in *pagecontent*. Initially, when an article is created, an empty tuple is inserted in *page*.
- **retLatest.** Retrieving the latest article works in less time as no join must be performed anymore. The content attribute of *page* can be read directly.
- **retFirst.** The downside when retrieving the first article is that it has to be computed out of all differences in *pagecontent*, starting from the latest version of *page*. This operation has linear complexity with the number of edits.

As we use the version control software *Git* as our competitor, we store all pages on the file system with one file per page and one commit per version. Traversing the predecessors of a commit works efficiently in *Git*. We store every article on a separate branch to avoid long chains when retrieving an earlier version.

- **insert.** For every new page, we create a new branch with the same name as the article descending from an empty commit. Every commit of a branch represents a page edit.
- **retLatest.** Retrieving the latest version is identical to a look up to the latest commit of the corresponding branch.
- **retFirst.** We retrieve the first version by traversing all commits to the origin. The time complexity is linear with the number of commits.

LiteTree allows branches to be created for each new version of a page. It stores every version uncompressed, which allows any arbitrary article to be retrieved efficiently. We rely on the internal versioning mechanisms and use only one relation *page* for the content.

- **insert.** A page edit is translated to an SQL update statement and a page creation to an SQL insert statement.
- **retLatest/retFirst.** As the commits are enumerated for every branch, we can access and retrieve any commit.

9.6 Versioning Data Type in SQL

To support text versioning within the Umbra database system, this section introduces a versioning data type that can be used as an SQL attribute to store multiple versions of a text in a tuple (see Listing 9.13). This data type is designed for maximum throughput, as it stores the most recent string as a whole and calculates previous versions using the differences to the latest version (*backward deltas*). This section first explains the concept and the storage layout for the versioning data type in Umbra as implemented by Karnowski [66] before introducing the operations exposed in SQL to insert, to update and to retrieve a certain version.

```
1 CREATE TABLE wikidiff (title text, content difftext);
2 INSERT INTO wikidiff (SELECT 'example', BUILD('one', 'one␣new␣version', 'one␣old␣version'));
3 SELECT GET_CURRENT_VERSION(difftext) FROM wikidiff;
```

LISTING 9.13: Proposed data type `DiffText` for text versioning.

9.6.1 Umbra Integration

The goal is to store the latest version as a snapshot followed by backward deltas to restore older versions. Each delta consists of the actual patch, a text string, and the range where to insert the patch. For example, when updating the text string "*one new version*" to "*one old version*", we store the latest version as a whole and, to retrieve the previous one, the patch "*new*" to insert at $[4, 8)$ (we start counting by 0).

Umbra provides a flexible-size string data type, called *RuntimeString*, from which all SQL attribute types with a size of more than 128 Bit are derived. Our versioning data type is based on this data type for variable length data as it uses the allocated memory for an internal representation as a `struct` in `C` (see Listing 9.14). The internal representation stores the offset and length of the current version stored as a snapshot to avoid look-ups for this particular and most common use-case (line `2`/`3`). The internal representation consists further of an array of deltas, each of which contains the offset relative to the previous patch, the start and end positions of the characters to be patched, and a tag indicating whether the string contains a full snapshot instead of a patch.

```
1  struct DiffTextRepresentation {
2     uint32_t currentOffset;     // Offset of current version in data section
3     uint32_t currentLength;     // Length of current version
4     uint32_t arraySize;         // The size of the version pointer's array
5     uint16_t diffsToFullCount; // Counter of diffs until next full version
6     struct {                    // Array of pairs, pointing into the data section
7        uint32_t offset;         // Offset of version in data section
8        bool full;               // Is this a full version?
9        uint32_t patchStart;     // Start of patch
10       uint32_t patchEnd;       // End of patch
11    } versionPointers[];
12    // Data section follows this struct immediately
13 };
```

LISTING 9.14: Data layout for the versioning data type [66].

In addition to the most recent version, we store snapshots of versions in-between the patches for linear retrieval time complexity. Figure 9.7 shows an example of five versions, where a snapshot is stored for every third version. So the third and currently the fifth version are stored as a snapshot (T_3, T_5) and the first, second and fourth version have to be restored from patches ($D_{T_2 \to T_1}$, $D_{T_3 \to T_2}$, $D_{T_5 \to T_4}$).

$D_{T_2 \to T_1}$	$D_{T_3 \to T_2}$	$\mathbf{T_3}$	$D_{T_5 \to T_4}$	$\mathbf{T_5}$

FIGURE 9.7: Five versions with every third version stored as a complete snapshot (bold) [66].

9.6.2 SQL Integration

To query the versioning data type within an SQL query, we offer three operations to create or to add a version and two operations plus one table operator to retrieve one or multiple versions.

- `BUILD(`$T_1, \ldots, T_N$`)` creates a new object given N versions.
- `APPEND(`$D, T_1, \ldots, T_N$`)` appends N versions to an existing object D.
- `SET_CURRENT_VERSION(`D`,` T`)` adds a new version T to an existing object D, equal to `APPEND(`D, T`)`.
- `GET_VERSION_BY_ID(`D, N`)` extracts version number N from object D.
- `GET_CURRENT_VERSION(`D`)` returns the latest version of object D. When N is the total number of versions, this call is equal to `GET_VERSION_BY_ID(`D, N`)`.
- `EXPAND(`D, M, N`)` is a unary database table operator that extracts the versions M to N.

Figure 9.8 shows the internal storage representation after three versions of a text string have been inserted (see Listing 9.15). First, the object is created using the text string "*one*", so no delta exists (`arraySize=0`) and the offset points to the beginning (`currentOffset=0`) of the data. Afterwards, "␣*new*␣*version*" is added to the string. The delta now contains the information on how to restore the previous version, so the offset for the patch (`0` as no text needs to be added) and the range $[3, 15)$ to omit. Finally, we change the string to "*one*␣*old*␣*version*", which becomes the current version stored as a snapshot. So the added delta to retrieve the previous version contains the range $[4, 7)$ to be replaced and the patch "*new*" given by its end position 3, its start position is implicitly given by the end of the previous one (0).

```
1 CREATE TABLE example (value DiffText);
2 INSERT INTO example (SELECT BUILD('one'));
3 UPDATE example SET value=SET_CURRENT_VERSION(value,'one new version');
4 UPDATE example SET value=SET_CURRENT_VERSION(value,'one old version');
```

LISTING 9.15: Exemplary insert history.

currentOffset	0
currentLength	3
arraySize	0
diffsToFullCount	0
Data	one

(A) Only one version: "*one*"

currentOffset	0
currentLength	15
arraySize	1
diffsToFullCount	1
offset	0
full	*false*
patchStart	3
patchEnd	15
Data	one new Version

(B) Modification to "*one new version*".

currentOffset	3
currentLength	15
arraySize	2
diffsToFullCount	2
offset	0
full	*false*
patchStart	3
patchEnd	15
offset	3
full	*false*
patchStart	4
patchEnd	7
Data	newone old version

(C) Modification to "*one old version*".

FIGURE 9.8: Internal representation [66] of the versioning data type after inserting: (a) "*one*", (b) "*one new version*" and (c) "*one old version*".

9.7 Evaluation

The evaluation section discusses the benefits and obstacles arising from the different storage approaches with respect to the different benchmark operations. Therefore, we evaluate the results of *MusaeusDB*, the *TardisBenchmark* and the performance of the versioning data type.

9.7.1 MusaeusDB

To evaluate *MusaeusDB*, we have chosen two settings: In order to measure the impact of extending versioning to support multiple tables, we have reimplemented versioning on single datasets as described for *OrpheusDB*, but using C++ instead of Python to be compiled

instead of interpreted. 10^6 tuples of fictional users and corresponding tasks served as test data. We first initialised the schema by adding the *rid* and a version table, then we performed a checkout, updated every tuple and committed the changes. For the second setup, we compared the speed up gained from the main-memory database system *HyPer* [69] in contrast to *PostgreSQL* as the underlying database system. We performed the test on a Debian 9 machine with four cores of Intel i7-7700HQ CPU, running at 2.80 GHz clock frequency each, and 16 GiB RAM. As we see in Figure 9.9, *MusaeusDB* needed twice as much the time as *OrpheusDB*, as it has to keep track of two tables for each version. When *HyPer* was used as the underlying database system instead of *PostgreSQL*, we only needed less than half the amount of time, in average.

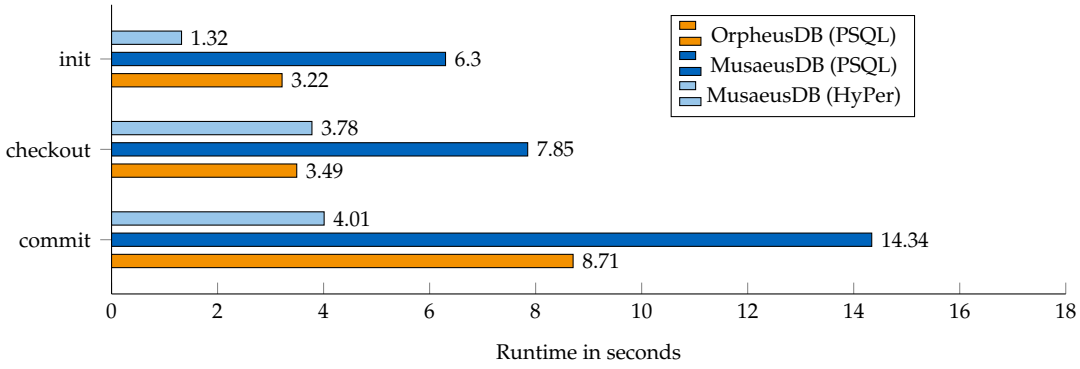


FIGURE 9.9: Runtime of *MusaeusDB* in contrast to *OrpheusDB* and in dependency of the underlying database system, *PostgreSQL* (PSQL) or *HyPer*.

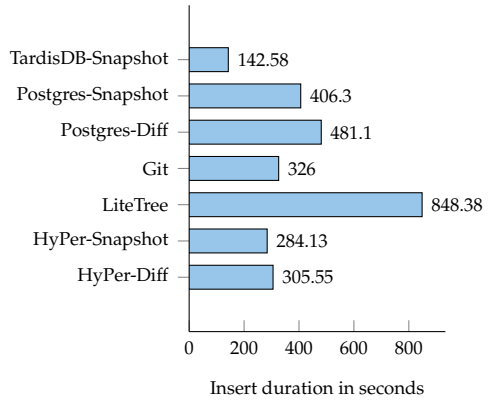
9.7.2 TardisBenchmark

This section discusses the runtime of the four storage approaches broken down into the different benchmark operations. The *TardisBenchmark* was run on an Ubuntu 18.04 server with an Intel Xeon CPU E5-2660 v2 with 2.20 GHz (20 cores) and 256 GiB DDR4 RAM. We used 2 TB of SSD storage to get the best performance out of the disk-based database systems SQLite and PostgreSQL. The benchmark was run single-threaded. We refer to the execution times measured by Schmeißer [110] (for TardisDB) and Karnowski [65].

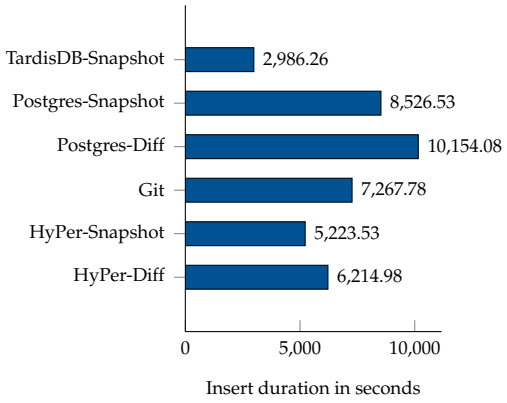
Insert

We inserted the Wikipedia dumps with full page edit history[6] from 1 August, 2018 for pages 10 up to 2,087 using the four storage approaches presented. The pages were stored as snapshots or as differences in the database systems *PostgreSQL* version 10.5 (classical disk-oriented) and *HyPer* (in main-memory). The uncompressed XML dump file takes up to 76.8 GB of space. As the dump consists mostly of text, nearly the same amount of text was inserted into the database. To measure the performance of the operations for smaller datasets, we also ran our benchmark with only 3.7 GB of data (pages 30,227 to 30,303).

As we see in Figure 9.10, storing snapshots is always faster than computing the differences beforehand. That is valid for both *PostgreSQL* and *HyPer*, where the latter outperforms the disk-based database system, as no tuples have to be written to disk. *LiteTree* (see Figure 9.10a) shows the worst insert performance. When performing on the larger dataset (see Figure 9.10b), *Git* shows better runtime than *PostgreSQL*, but worse than *HyPer*, whereas *SQLite* is not capable of managing more than 1,024 branches at all. *TardisDB*—as optimised to this setting—was the fastest system.

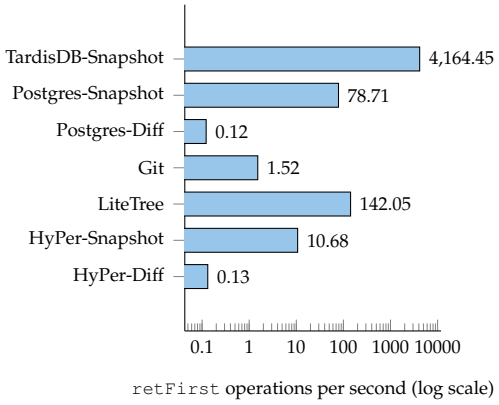


(A) `insert`: small (3.7 GB) dataset [65, 110].

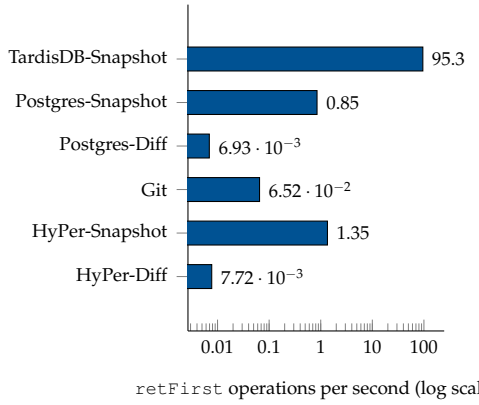


(B) `insert`: larger (76.8 GB) dataset [65, 110].

FIGURE 9.10: Benchmark results for the `insert` operation: The left side shows the insertion of the small dataset, the right that of the larger. LiteTree was not capable of handling enough branches for the larger dataset.

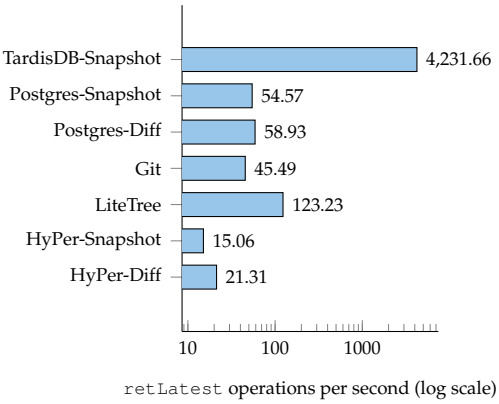


(A) `retFirst`: small (3.7 GB) dataset [65, 110].

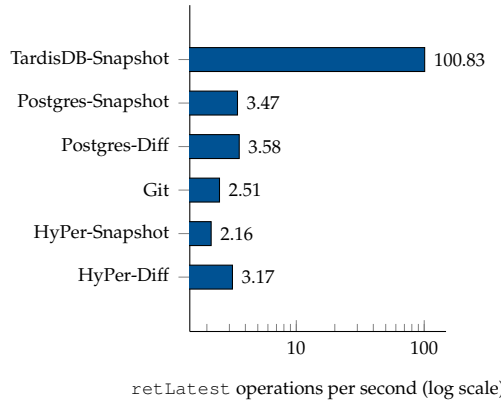


(B) `retFirst`: larger (76.8 GB) dataset [65, 110].

FIGURE 9.11: Benchmark results for the `retFirst` operation: small (left) and larger (right) dataset. When pages are stored as snapshots, any version can be accessed immediately. When the differences between versions are stored instead, the first version has to be patched out of all deltas, which results in runtimes up to 50 times slower.



(A) `retLatest`: small (3.7 GB) dataset [65, 110].



(B) `retLatest`: larger (76.8 GB) dataset [65, 110].

FIGURE 9.12: Benchmark results for `retLatest` operation with small (left) and larger (right) dataset. All storage approaches show about the same page retrieval time, as the latest version is always stored as a whole. The difference-based storage layout shows slightly better performance as no joins are performed.

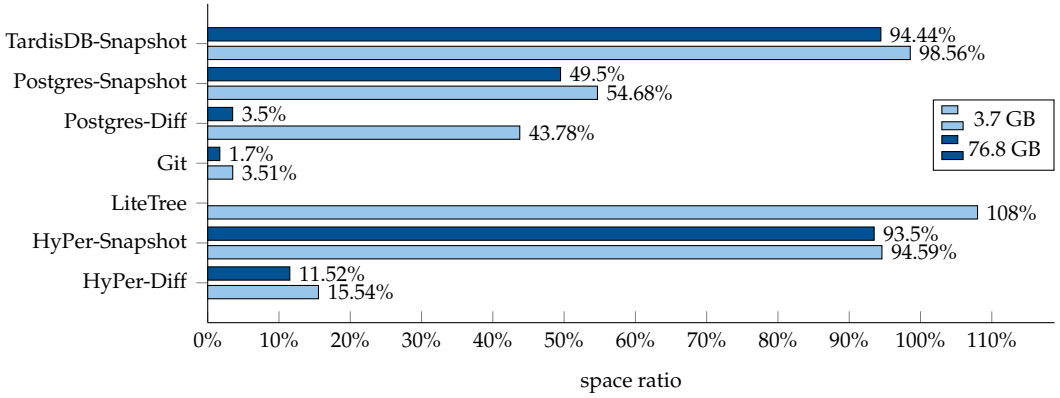


FIGURE 9.13: Relative space consumption to the original size (numbers taken from [65, 110]).

Space Consumption

After inserting the pages, we analysed the space consumption in comparison to the original size of the two datasets (see Figure 9.13). The most space-consuming approaches—as no compression took place and the whole snapshots were copied—are *LiteTree*, *HyPer-Snapshot* and *TardisDB-Snapshot*.

LiteTree stores each page as a new database entry on each commit and creates no index structures. As it just copies the pages, its insertion time was one of the lowest ones, but also space consuming with more than the initial 3.7 GB. *HyPer* does not compress text-based data, so it needed the same amount of storage but in main-memory when storing each snapshot. One improvement would be to compress the articles manually before insertion, with the drawback of a higher runtime. Unexpectedly, *PostgreSQL* needed about 38 GB for the large dataset in both approaches (*snapshot* and *diff*), as it did not delete tuples immediately. After a cleanup, the database size for the *Postgres-Diff* had been shrunk to 2.8 GB for the large and 1.62 GB for the small dataset.

Git initially needed about 38 GB—the same amount of memory as *PostgreSQL*. The benchmark used `libGit` for insertion [65], so the pages were stored uncompressed. After we manually ran the garbage collector *git gc*, the storage size shrank to 1.3 GB for the large dataset and 130 MB for the small one, so 1.7 % (3.51 % respectively) of the size of the uncompressed XML file. Here, as in all difference-based approaches, the bigger the dataset, the higher the compression rate. The size was reduced due to packing, the process of compressing the *Git* repository, implicitly called by the garbage collector. Packing the *Git* repository results in the lowest database size in our setup.

Retrieve First

Retrieving the first version works faster when the page content can be accessed as snapshots directly. This can be seen using either a main-memory or a classical database system as data storage for small (see Figure 9.11a) or large workloads (see Figure 9.11b). *Git* was always faster than the difference-based storage layout using its own delta compression. The performance of *PostgreSQL* decreased with the number of inserted elements: while it performed the fastest after *LiteTree* for the small load, the runtime decreased when the larger dataset was handled. *LiteTree* showed a good performance as every commit in a branch has an incrementally-growing number. To fetch the first version, we just specify it by `article-branch.1` [65]. Also in this setting, the architecture of *TardisDB* allowed the greatest number of retrieving operations per second, but this number decreases with higher workload, as a longer version chain has to be traversed.

[6] `https://dumps.wikimedia.org/enwiki/20180801/`

Retrieve Latest

As we see in Figure 9.12, our difference-based storage layout always showed the best performance except for the small workload (see Figure 9.12a), where *LiteTree* performed the fastest. The reason that this outperformed the snapshot-based storage layout was that the joins were eliminated. For the larger workload (see Figure 9.12b), we see that the database systems outperformed *Git* by 20 % of its runtime. As *TardisDB* was tuned for fast scans on the latest tuples, it allowed the most read transactions independent of the workload size.

9.7.3 Versioning Data Type

This section presents the evaluation of the versioning data type in Umbra. The Wikipedia dumps of 1 September, 2019 (pages 971896 to 972009), which contain the full page edit history, serve as data. The size of the corresponding XML file is about 120 MB. We benchmark on an Ubuntu 18.04 LTS server with an Intel Xeon CPU E5-2660 v2 with 2.20 GHz (20 cores) and 256 GiB DDR4 RAM.

We first measure the size required when varying the frequency X of how often to save a complete snapshot and not just the patch (see Figure 9.14). Saving every version as a snapshot ($X = 1$) requires about 62 MB, this is the bare text size of all contained articles. We can reduce the memory consumption to 12 % of the original size when storing every 10,000th version as a snapshot and the remaining versions as patches in-between. So a compression rate of about 88 % is achieved for this data dump by just storing the patches. It can be observed that a good compression is already achieved when storing every 20th version as a snapshot while limiting the maximum number of patches to 20.

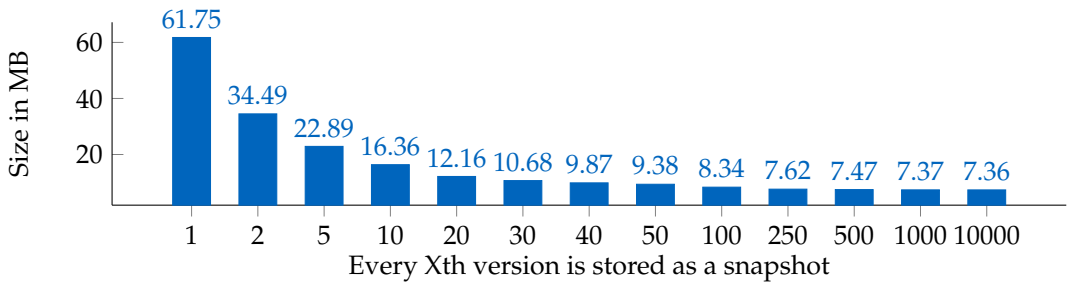


FIGURE 9.14: Memory consumption depending on the frequency of stored snapshots [66].

We now consider a frequency of saving every 50th article ($X = 50$) as a snapshot to evaluate the data type's performance. We compare the runtime and memory consumption of storing an article's history in one tuple using the versioning data type (*diff*, see Listing 9.16) to one tuple per version based on a text data type (*snapshot*, see Listing 9.17). Figure 9.15a shows the ratio for the memory consumption of the three parts of the underlying struct (patches, snapshots, and the headers including the array for the pointers to the data): At a frequency of $X = 50$, less than a third of the memory is used for the snapshots, while most of the memory is consumed by the patches. So most of the memory is used for patches containing only the differences, and thus, avoiding redundancy.

```
1 INSERT INTO t (text)
2   VALUES (BUILD(T1, ..., TN));
3 SELECT EXPAND(text, 1, N) from t;
```

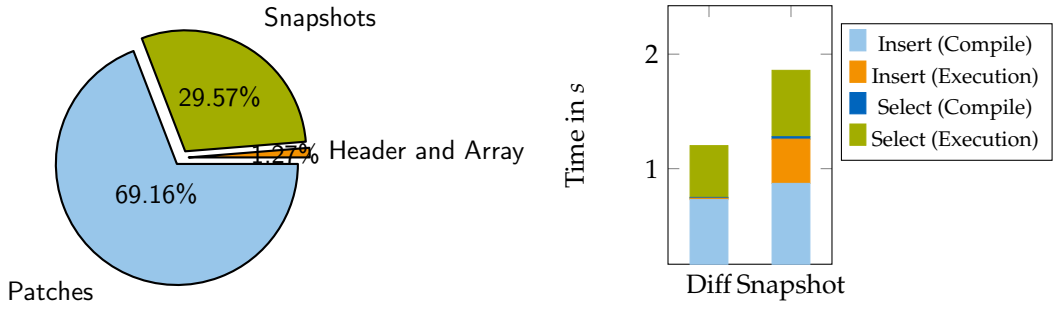
LISTING 9.16: Diff [66].

```
1 INSERT INTO t (rev_id, text)
2   VALUES (1, T1), ..., (N, TN);
3 SELECT text from t;
```

LISTING 9.17: Snapshot [66]

Figure 9.15b shows the runtime for the two approaches consisting of the insertion and retrieval time, broken down into compile and execution time. First, the time for inserting all versions of all articles in the considered Wikipedia dump is measured, then the time for retrieving all articles. In both approaches, the compile time for insertion dominates, since the parsing time for text input is included. Our versioning data type is faster at insertion and retrieval because fewer operations are required. To summarise, the versioning data type

consumes less memory as it stores mostly patches, while having good retrieval times, since fewer operations are required.



(A) Memory Ratio (B) Runtime

FIGURE 9.15: (a) Memory ratio of the individual parts of the underlying struct and (b) runtime comparison to naive approach of storing every version as a tuple [66].

9.8 Conclusion

This chapter showed how to adapt versioning for database systems including an SQL extension and evaluated the performance and space consumption of storing the differences between pages in contrast to storing the whole snapshots. We first developed a database schema, which was capable of managing versions over multiple relations, and described *MusaeusDB* as a tool to deal with the schema.

To include versioning inside modern main-memory database systems, we adapted multi-version concurrency control for our prototype *TardisDB* that generated low-level machine code. It was equipped with a modified table scan operator with bitmaps to indicate the affiliation of tuples to branches and a version chain to track their modification history. As tuples and branches came with a timestamp each, the table scan operator could verify in a version bitmap for each tuple whether that tuple is contained in the current branch. The corresponding SQL extension supported named branches over multiple tables, which comprises a statement for branch creation and an auxiliary keyword after each table to determine the branch.

TardisDB performed best at retrieving the latest tuples, the setup for which it was designed. The extension did not slow down the read throughput on the master branch and retrieved other versions faster than comparable systems. Our *TardisBenchmark* evaluated storing deltas against storing snapshots of pages based on the schema and data of MediaWiki. Our benchmark showed that it was indeed possible to shrink the amount of storage used by a factor of ten, while still offering comparable time when the pages are demanded. Using delta compression inside of the Umbra database system, we implemented a data type that compresses multiple versions of a text string. Using this data type for Wikipedia articles, we achieved a compression rate of up to 11.9 % and outperformed the traditional text data type, when storing each version as one tuple individually, by an order of magnitude.

Overall, we showed versioning techniques that fit for database systems and proved the capability, with Wikipedia as a real world use case. With the developed web interface, we aimed to demonstrate the simplicity of extending SQL for versioning, which should also increase the acceptance of SQL for further tasks such as version control. To support further research on database versioning besides our proposed storage techniques, we provide our *TardisBenchmark*, which is capable of handling a flexible-sized workload.

Chapter 10

Conclusion and Future Work

This chapter sums up the findings of dataset versioning, in-database machine learning and analytics, and concludes by providing an outlook on standardisations required to run the presented algorithms within other database systems or to allow their extension for further operators.

10.1 Conclusion

This thesis has extended SQL within modern database systems to support data analysis, gradient descent for machine learning and dataset versioning. To express data analysis in SQL, we presented algorithms within a scripting language and recursive tables that require aggregate or window functions. We showed that their performance within a modern database system such as HyPer or Umbra was comparable to the runtime within an extended traditional database system, i.e., PostgreSQL with MADlib and PL/pgSQL. Recursive tables with support for aggregate functions are also capable of expressing gradient descent, which allowed the elimination of the extensive ETL process for model training. To further facilitate the computation of gradient descent, we integrated automatic differentiation as an operator in SQL, which accelerated the runtime by the number of cached expressions. Automatic differentiation was based on an SQL lambda function. This allowed arbitrary expressions on SQL data types including arrays interpreted as matrices to serve as a loss function for gradient descent. For further acceleration, we included automatic differentiation within a dedicated operator for gradient descent, which allowed off-loading training to GPU. For training on GPUs, we developed fine-grained learners, which allowed the highest possible throughput even for small batch sizes. When using these extensions for a translation of a meta-language for machine learning to SQL, we figured out that most operations transformed the data into arrays before processing. Therefore, we extended the code-generating database system Umbra by an interface for arrays turning it into an array database system with in-memory performance.

Dataset versioning on top of database systems was extended to support multiple tables per version and benefitted from the performance increase of an in-memory database system, i.e., HyPer. To fully incorporate dataset versioning inside of a main-memory engine, we developed TardisDB, a database system whose table scan operator supported versioning based on branch bitmaps and on multi-version concurrency control for tracking each tuple's history. We proposed a downward compatible SQL extension to query and to address different branches. Based on Wikipedia as a use case, the evaluation showed that TardisDB performed the best for retrieving the latest version of an article, whereas the memory consumption would be reduced further if text compression was used.

10.2 Future Work

When benchmarking dataset versioning, delta text compression minimised the memory consumption of Wikipedia articles significantly. Text compression could be combined with multi-version concurrency control, as used in TardisDB, to further reduce the space consumption of a database system with incorporated dataset versioning. Besides, TardisDB supports only one client per database instance. However, in a production setup, a database server should handle multiple clients. So the interplay of versioning and multi-version concurrency control will be of interest and should be enabled.

Support for the SQL standard varied from system to system, making it difficult to compare database systems for data analysis. To enable the interchangeability of database systems, we consider a homogeneous implementation of the SQL standard to be essential. For example, aggregate and window functions within recursive tables should be supported within common database systems. Moreover, HyPer's data mining operators expect tables as input arguments. These so-called polymorphic table functions are part of the SQL:2016 standard [97]. This standard should be implemented in other database systems [118] in order to enable automatic differentiation and gradient descent as an operator. Furthermore, an array data type, such as that provided by PostgreSQL, was used as the basis for matrix operations. Such a data type would be helpful also within other database systems and should be included in a future SQL standard.

As an alternative to an array data type, this thesis has shown that database tables can be interpreted as arrays using a separate interface. Instead of providing an ArrayQL interface, applications can translate array expressions into relational algebra directly. As an example, the presented declarative meta-language *MLearn* can be extended to perform linear algebra on SQL tables instead of on the array data type. Another approach would concern the tuples in their materialised form. Instead of using relational algebra, an array interface could access the persisted data. This is needed when a tabular representation for array data should be chosen instead of a relational one.

The presented in-database GPU support was limited to gradient descent within a dedicated operator only. Instead of a dedicated operator, GPU support can accelerate operations on specific data types or operators of the relational algebra. To reduce the amount of transferred data, entire operator plans of a code-generating database system should be compiled together and off-loaded to GPU. For matrix algebra on the array data type, successive operations can be merged to transfer the data only once.

Appendix A

SQL for Data Analysis: Queries

A.1 HyPer

```
-- k-means algorithm as hyperscript
create schema if not exists kmeansscript;
create table kmeansscript.points (id varchar(8), x float, y float);
create table kmeansscript.clusters (cid integer, x float, y float, points integer);
create table kmeansscript.pointsclusters (cid integer, pid varchar(8));

-- some sample data
create or replace function kmeansscript.insertSampleIntoPoints() as $$
   debug_print('insert_sample_points');
   insert into kmeansscript.points(VALUES ('A',2,12),('B',3,11),('C',3,8),('D',5,4),('E',7,5),('F',7,3),('G',10,8),('H',13,8));
$$ language 'hyperscript' strict;

-- create random points
create or replace function kmeansscript.random(count integer not null, maxx integer not null, maxy integer not null) returns integer
      not null as $$
   debug_print('insert_random_points:',count);
   delete from kmeansscript.points where true;
   select index as i from sequence(0,count){
      insert into kmeansscript.points(values(i, random(maxx), random(maxy)));
   }
   return 0;
$$ language 'hyperscript' strict;

-- the iterate function, returns 0 on fixpoint or number of chnaged rows
create or replace function kmeansscript.iterate() returns integer not null as $$
   debug_print('iterate');
   table clusters_tmp (cid integer, x float, y float, points integer);
   insert into clusters_tmp (
      select cid, avg(px), avg(py), count(*) from (
        select cid, p.x as px, p.y as py, rank() OVER (
          partition by p.id
          order by (p.x-c.x)*(p.x-c.x)+(p.y-c.y)*(p.y-c.y) asc,
            (c.x*c.x+c.y*c.y) asc)
        from kmeansscript.points p, kmeansscript.clusters c
      ) x
      where x.rank=1
      group by cid
   );
   select count(*) as c_count from (select * from kmeansscript.clusters except select * from clusters_tmp);
   delete from kmeansscript.clusters where true; -- move tmp to main cluster table
   insert into kmeansscript.clusters (select * from clusters_tmp);
   return c_count; -- return number of different rows
$$ language 'hyperscript' strict;

-- finally, combine points to clusters
create or replace function kmeansscript.pointsToClusters() as $$
   insert into kmeansscript.pointsclusters(
      select cid,pid from (
          select cid, p.id as pid, rank() OVER (
              partition by p.id
              order by (p.x-c.x)*(p.x-c.x)+(p.y-c.y)*(p.y-c.y) asc,
                  (c.x*c.x+c.y*c.y) asc)
          from kmeansscript.points p, kmeansscript.clusters c
      ) x
      where x.rank=1
   );
$$ language 'hyperscript' strict;

-- main method to manage iteration
create or replace function kmeansscript.run(centers integer not null) as $$
   debug_print('run_kmeansscript;_centers:', centers);
   delete from kmeansscript.clusters where true;
   insert into kmeansscript.clusters(
      select id,x,y,0 from (select *,rank() over (order by id) from kmeansscript.points) where rank <= centers
   );
   select index as i from sequence(0,100) {
      select kmeansscript.iterate() as c_count;
         if(c_count = 0){
            break;
         }
   }
   select kmeansscript.pointsToClusters();
$$ language 'hyperscript' strict;

create or replace function kmeansscript.benchmark() as $$
   debug_print('benchmark');
   select kmeansscript.random(1000, 1000, 1000);
   select kmeansscript.run(5);
$$ language 'hyperscript' strict;
```

LISTING A.1: k-Means.

```
-- dbscan algorithm as hyperscript
create schema if not exists dbscanscript;
create table dbscanscript.points (id varchar(8), x float, y float);
create table dbscanscript.pointsclustered (id varchar(8), x float, y float, clusterid integer, noise boolean);

-- some sample data
create or replace function dbscanscript.insertSampleIntoPoints()
as $$
   debug_print('insert sample points');
   insert into dbscanscript.points(VALUES('A',2,12),('B',3,11),('C',3,8),('D',5,4),('E',7,5),('F',7,3),('G',10,8),('H',13,8));
$$ language 'hyperscript' strict;

-- create random points
create or replace function dbscanscript.random(count integer not null, maxx integer not null, maxy integer not null) returns integer
      not null as $$
   debug_print('insert random points:',count);
   delete from dbscanscript.points where true;
   select index as i from sequence(0,count){
      insert into dbscanscript.points(values(i, random(maxx), random(maxy)));
   }
   return 0;
$$ language 'hyperscript' strict;

create or replace function dbscanscript.insertCluster(clusterid integer not null, eps float not null, max integer not null) returns
      integer not null as $$
   table pointsTmp (id varchar(8), x float, y float);
   insert into pointsTmp(
         (select * from dbscanscript.points except (select id,x,y from dbscanscript.pointsclustered) LIMIT 1)
   );
   select count(*) as ret from pointsTmp;
   select index from sequence(1,max){
      select count(*) as newc from pointsTmp c, dbscanscript.points p
         where pow(p.x-c.x,2)+pow(p.y-c.y,2)<pow(eps,2) and c.id<>p.id;
      insert into pointsTmp(
         select * from (select * from dbscanscript.points except select * from pointsTmp) c
            where exists (select * from pointsTmp p where pow(p.x-c.x,2)+pow(p.y-c.y,2)<pow(eps,2) and c.id<>p.id)
      );
      if(newc=0){
         break;
      }
      ret=ret+newc;
   }
   insert into dbscanscript.pointsclustered(select *,clusterid, false from pointsTmp);
   return ret;
$$ language 'hyperscript' strict;

-- main method to manage iteration
create or replace function dbscanscript.run(eps float not null, minPoints integer not null, maxiter integer not null) as $$
   delete from dbscanscript.pointsClustered where true;
   insert into dbscanscript.pointsClustered(
      select *,null,true from dbscanscript.points np where minPoints>(
         select count(*) from dbscanscript.points p where pow(p.x-np.x,2)+pow(p.y-np.y,2)<pow(eps,2)
      )
   );
   select index from sequence (1,maxiter){
      select dbscanscript.insertCluster(index,eps,maxiter) as cond;
      if(cond=0){
         break;
      }
   }
$$ language 'hyperscript' strict;

create or replace function dbscanscript.benchmark() as $$
   debug_print('benchmark');
   select dbscanscript.random(10,10,10);
   select dbscanscript.run(3,2,10);
$$ language 'hyperscript' strict;
```

LISTING A.2: DBSCAN.

```
-- pagerank algorithm as hyperscript
create schema if not exists prscript;
create table prscript.edges        (src Integer, dst Integer);
create table prscript.pagerank     (Node Integer, Pagerank Float);
create table prscript.pagerank_tmp (Node Integer, Pagerank Float);

-- some sample data
create or replace function prscript.insertSampleIntoEdges() as $$
insert into prscript.edges(VALUES (1,2),(1,3),(2,3),(3,4),(4,1));
$$ language 'hyperscript' strict;

-- create random points
create or replace function prscript.random(nodes integer not null, edges integer not null) returns integer not null as $$
   debug_print('insert random edges; edges:', edges, '; nodes:', nodes);
   delete from prscript.edges where true;
   select index as i from sequence(0,edges){
      insert into prscript.edges(values(random(nodes), random(nodes)));
   }
   return edges;
$$ language 'hyperscript' strict;

-- the iterate function, returns 0 on fixpoint or number of chnaged rows
create or replace function prscript.iterate(alpha float not null) returns integer not null as $$
   table pagerank_tmp(Node Integer, Pagerank Float);
   insert into pagerank_tmp (
      select dst, alpha*(CAST ((select count(*) from prscript.pagerank) AS FLOAT))+ (1-alpha)*sum(Beitrag)
      from (
         select e.dst, p.Pagerank /(select count (*) from prscript.edges x where x.src=e.src) as Beitrag
         from prscript.edges e , prscript.pagerank p
         where e.src=p.Node
      ) i
      group by dst
   );
   select count(*) as c_count from (select * from prscript.pagerank except select * from pagerank_tmp);
   delete from prscript.pagerank where true; -- move tmp to main cluster table
   insert into prscript.pagerank (select * from pagerank_tmp);
   return c_count; -- return number of different rows
$$ language 'hyperscript' strict;

-- main method to manage iteration
create or replace function prscript.run(alpha float not null) as $$
   delete from prscript.pagerank where true;
   insert into prscript.pagerank(
      select distinct e.src, 1/(CAST ((select count(distinct src) from prscript.edges) AS FLOAT))
      FROM prscript.edges e
   );
   select index as i from sequence(0,100) {
      select prscript.iterate(alpha) as c_count;
         if(c_count = 0){
            break;
         }
   }
$$ language 'hyperscript' strict;
```

LISTING A.3: PageRank.

```
create schema if not exists aprioriscript;
create table aprioriscript.sales (tid integer, item integer);
create table aprioriscript.fis (item integer[]);

-- create random points
create or replace function aprioriscript.random(items integer not null, buckets integer not null, maxbucketsize integer not null)
       returns integer not null as $$
   debug_print('create randombuckets; items:', items, '; buckets:', buckets, 'maxBucketSize:', maxbucketsize);
   select index as bucket from sequence(0,buckets){
      -- take a random bucket size between 0 and maxbucketsize
      select index as size from sequence(0,random(maxbucketsize)){
         insert into aprioriscript.sales values (bucket,random(random(random(random(random(random(items))))));
      }
   }
   select count(*) as ret from aprioriscript.sales;
   return ret;
$$ language 'hyperscript' strict;

create or replace function aprioriscript.findFI(minsupp integer not null) as $$
   debug_print('find frequent itemsets; minsupp (abs):', minsupp);
   with recursive
   transactions (tid, bucket) as(
      select tid, array_agg(item) from aprioriscript.sales group by tid
   ),
   sales_supp as (
     select item
     from aprioriscript.sales
     group by item
     having count(*) >= minsupp
   ), --items with minSupp >= 2
   frequentitemsets as (
         (select distinct array[p.item]::integer[] as items from sales_supp p)
      union all (
         -- apriori-gen()
        select distinct array_append(t.items,p.item::integer)::integer[]
        from frequentitemsets t, sales_supp p
        where (
           select count(*)
           from transactions t2
           where array_append(t.items,p.item::integer)::integer[] <@ ( t2.bucket )
        ) >= minsupp and
           t.items[(select count(*) from unnest(t.items))]<p.item
     )
   )
   insert into aprioriscript.fis (select * from frequentitemsets);
$$ language 'hyperscript' strict;

create or replace function aprioriscript.benchmark() returns integer not null as $$
   debug_print('run benchmark');
   select aprioriscript.random(1000,100000,80);
   select aprioriscript.findFI(10000);
   select count(*) as cs from aprioriscript.fis;
   return cs;
$$ language 'hyperscript' strict;
```

LISTING A.4: Apriori.

A.2 PostgreSQL

```
create schema if not exists kmeansscript;
create table kmeansscript.points (id integer, x float, y float);
create table kmeansscript.clusters (cid integer, x float, y float, points integer);
create table kmeansscript.clusters_tmp (cid integer, x float, y float, points integer);
create table kmeansscript.pointsclusters (cid integer, pid integer);

create or replace function kmeansscript.random(count integer, maxx integer, maxy integer) returns integer as $$ BEGIN
   delete from kmeansscript.points where true;
   FOR i IN 1..count LOOP
      insert into kmeansscript.points(values(i, random()*maxx, random()*maxy));
   END LOOP;
   return 0;
END $$ LANGUAGE plpgsql;

-- the iterate function, returns 0 on fixpoint or number of chnaged rows
create or replace function kmeansscript.iterate() returns integer as $$ BEGIN
   delete from kmeansscript.clusters_tmp where true;
   insert into kmeansscript.clusters_tmp (
      select cid, avg(px), avg(py), count(*) from (
        select cid, p.x as px, p.y as py, rank() OVER (
          partition by p.id
          order by (p.x-c.x)*(p.x-c.x)+(p.y-c.y)*(p.y-c.y) asc,
            (c.x*c.x+c.y*c.y) asc)
        from kmeansscript.points p, kmeansscript.clusters c
      ) x
      where x.rank=1
      group by cid
   );
   perform count(*) as c_count from (select * from kmeansscript.clusters except select * from kmeansscript.clusters_tmp) tmp;
   delete from kmeansscript.clusters where true; -- move tmp to main cluster table
   insert into kmeansscript.clusters (select * from kmeansscript.clusters_tmp);
   return 1;
END $$ LANGUAGE plpgsql;

-- finally, combine points to clusters
create or replace function kmeansscript.pointsToClusters() returns integer as $$ BEGIN
   insert into kmeansscript.pointsclusters(
      select cid,pid from (
          select cid, p.id as pid, rank() OVER (
              partition by p.id
              order by (p.x-c.x)*(p.x-c.x)+(p.y-c.y)*(p.y-c.y) asc,
                  (c.x*c.x+c.y*c.y) asc)
          from kmeansscript.points p, kmeansscript.clusters c
      ) x
      where x.rank=1
   );
   return 1;
END $$ LANGUAGE plpgsql;

-- main method to manage iteration
create or replace function kmeansscript.run(centers integer) returns integer as $$ BEGIN
   delete from kmeansscript.clusters where true;
   insert into kmeansscript.clusters(
      select id,x,y,0 from (select *,rank() over (order by id) from kmeansscript.points) ranks where rank <= centers
   );
   FOR i IN 0..100 LOOP
      perform kmeansscript.iterate() as c_count;
   END LOOP;
   return kmeansscript.pointsToClusters();
END $$ LANGUAGE plpgsql;

create or replace function kmeansscript.benchmark() returns integer as $$ BEGIN
   perform kmeansscript.random(1000, 1000, 1000);
   perform kmeansscript.run(5);
   return 1;
END $$ LANGUAGE plpgsql;
```

LISTING A.5: k-Means.

```
create schema if not exists dbscanscript;
create table dbscanscript.points (id varchar(8), x float, y float);
create table dbscanscript.pointsclustered (id varchar(8), x float, y float, clusterid integer, noise boolean);
create table dbscanscript.pointsTmp (id varchar(8), x float, y float);

-- some sample data
create or replace function dbscanscript.insertSampleIntoPoints() returns integer as $$ BEGIN
   insert into dbscanscript.points(VALUES('A',2,12),('B',3,11),('C',3,8),('D',5,4),('E',7,5),('F',7,3),('G',10,8),('H',13,8));
   return 0;
END $$ language plpgsql;

-- create random points
create or replace function dbscanscript.random(count integer, maxx integer, maxy integer) returns integer as $$ BEGIN
   delete from dbscanscript.points where true;
   FOR i IN 1..count LOOP
      insert into dbscanscript.points(values(i, random()*maxx, random()*maxy));
   END LOOP;
   return 0;
END $$ language plpgsql;

create or replace function dbscanscript.insertCluster(clusterid integer, eps float, max integer) returns integer as $$
DECLARE
   ret int;
   newc int;
BEGIN
   delete from dbscanscript.pointstmp where true;
   insert into dbscanscript.pointsTmp(
         (select * from dbscanscript.points except (select id,x,y from dbscanscript.pointsclustered) LIMIT 1)
   );
   select count(*) from dbscanscript.pointstmp into ret;
   ret=0;
   FOR i IN 1..max LOOP
      select count(*) from dbscanscript.pointsTmp c, dbscanscript.points p
         where pow(p.x-c.x,2)+pow(p.y-c.y,2)<pow(eps,2) and c.id<>p.id into newc;
      insert into dbscanscript.pointsTmp(
         select * from (select * from dbscanscript.points except select * from dbscanscript.pointsTmp) c
            where exists (select * from dbscanscript.pointsTmp p where pow(p.x-c.x,2)+pow(p.y-c.y,2)<pow(eps,2) and c.id<>p.id)
      );
      ret=ret+newc;
   END LOOP;
   insert into dbscanscript.pointsclustered(select *,clusterid, false from dbscanscript.pointstmp);
   return ret;
 END $$ language plpgsql;

-- main method to manage iteration
create or replace function dbscanscript.run(eps float, minPoints integer, maxiter integer) returns integer as $$
DECLARE
   cond integer;
BEGIN
   delete from dbscanscript.pointsClustered where true;
   insert into dbscanscript.pointsClustered(
      select *,null,true from dbscanscript.points np where minPoints>(
         select count(*) from dbscanscript.points p where pow(p.x-np.x,2)+pow(p.y-np.y,2)<pow(eps,2)
      )
   );
   FOR index IN 1..maxiter LOOP
      select dbscanscript.insertCluster(index,eps,maxiter) into cond;
   END LOOP;
   return 1;
 END $$ language plpgsql;

create or replace function dbscanscript.benchmark() returns integer as $$ BEGIN
   perform dbscanscript.random(10,10,10);
   perform dbscanscript.run(3,2,10);
   return 0;
 END $$ language plpgsql;
```

LISTING A.6: DBSCAN.

```
create schema if not exists prscript;
create table prscript.edges        (VFrom Integer, VTo Integer);
create table prscript.pagerank     (Node Integer, Pagerank Float);
create table prscript.pagerank_tmp (Node Integer, Pagerank Float);

-- some sample data
create or replace function prscript.insertSampleIntoEdges() returns integer as $$ BEGIN
insert into prscript.edges(VALUES (1,2),(1,3),(2,3),(3,4),(4,1));
return 1;
END $$ LANGUAGE plpgsql;

-- create random points
create or replace function prscript.random(nodes integer, edges integer) returns integer as $$ BEGIN
   delete from prscript.edges where true;
   FOR i IN 1..edges LOOP
      insert into prscript.edges(values(random()*nodes, random()*nodes));
   END LOOP;
   return edges;
END $$ language plpgsql;

-- the iterate function, returns 0 on fixpoint or number of chnaged rows
create or replace function prscript.iterate(alpha float) returns integer as $$ BEGIN
   delete from prscript.pagerank_tmp where true;
   insert into prscript.pagerank_tmp (
      select VTo, alpha*(CAST ((select count(*) from prscript.pagerank) AS FLOAT))+ (1-alpha)*sum(Beitrag)
      from (
         select e.VTo, p.Pagerank /(select count (*) from prscript.edges x where x.VFrom=e.VFrom) as Beitrag
         from prscript.edges e , prscript.pagerank p
         where e.VFrom=p.Node
      ) i
      group by VTo
   );
   perform count(*) as c_count from (select * from prscript.pagerank except select * from prscript.pagerank_tmp) a;
   delete from prscript.pagerank where true; -- move tmp to main cluster table
   insert into prscript.pagerank (select * from prscript.pagerank_tmp);
   return 1;
 END $$ language plpgsql;

-- main method to manage iteration
create or replace function prscript.run(alpha float) returns integer as $$ BEGIN
   delete from prscript.pagerank where true;
   insert into prscript.pagerank(
      select distinct e.VFROM, 1/(CAST ((select count(distinct VFrom) from prscript.edges) AS FLOAT))
      FROM prscript.edges e
   );
   FOR i IN 1..100 LOOP
      perform prscript.iterate(alpha) as c_count;
   END LOOP;
   return 1;
END $$ language plpgsql;
```

LISTING A.7: PageRank.

```
create schema if not exists aprioriscript;

-- tables for shopping cart items
create table aprioriscript.sales (tid integer, item integer);
create table aprioriscript.fis (item integer[]);

-- create random points
create or replace function aprioriscript.random(items integer, buckets integer, maxbucketsize integer) returns integer as $$ BEGIN
   FOR bucket IN 0..buckets LOOP
      -- take a random bucket size between 0 and maxbucketsize
      FOR i IN 0..random()*maxbucketsize LOOP
         insert into aprioriscript.sales values (bucket,random()*random()*random()*random()*random()*items);
      END LOOP;
   END LOOP;
   return 1;
END $$  language plpgsql;

create or replace function aprioriscript.findFI(minsupp integer) returns integer as $$ BEGIN
   with recursive
   transactions (tid, bucket) as(
      select tid, array_agg(item) from aprioriscript.sales group by tid
   ),
   -- frequent items
   sales_supp as (
     select item
     from aprioriscript.sales
     group by item
     having count(*) >= minsupp
   ), --items with minSupp >= 2
   frequentitemsets as (
         (select distinct array[p.item]::integer[] as items from sales_supp p)
      union all (
         -- apriori-gen()
        select distinct array_append(t.items,p.item::integer)::integer[]
        from frequentitemsets t, sales_supp p
        where (
           -- count support
           select count(*)
           from transactions t2
           where array_append(t.items,p.item::integer)::integer[] <@ ( t2.bucket )
        ) >= minsupp and
           t.items[(select count(*) from unnest(t.items))]<p.item
     )
   )
   insert into aprioriscript.fis (select * from frequentitemsets);
   return 1;
END $$  language plpgsql;

create or replace function aprioriscript.benchmark() returns integer as $$ BEGIN
   perform aprioriscript.random(100,100,8);
   perform aprioriscript.findFI(100);
   perform count(*) as cs from aprioriscript.fis;
   return 1;
END $$  language plpgsql;
```

LISTING A.8: Apriori.

Appendix B

In-Database Machine Learning: Scripts

```
create table if not exists data (x1 float, x2 float, x3 float, x4 float, x5 float, x6 float, y1 float, y2 float, y3 float, y4 float,
      y5 float, y6 float);
insert into data (select *, 0.5+0.4*x1,
                            0.5+0.4*x1+0.25*x2,
                            0.5+0.4*x1+0.25*x2+0.1*x3,
                            0.5+0.4*x1+0.25*x2+0.1*x3+0.2*x4,
                            0.5+0.4*x1+0.25*x2+0.1*x3+0.2*x4+0.6*x5,
                            0.5+0.4*x1+0.25*x2+0.1*x3+0.2*x4+0.6*x5+0.7*x6
         from (select random() x1, random() x2, random() x3, random() x4, random() x5, random() x6 from generate_series(1,100000)));

\record gd.csv Manually,6,100000,0.1,50
with recursive gd (id, a1, a2, a3, a4, a5, a6, b) as (
   select 0,1::float,1::float,1::float,1::float,1::float,1::float,1::float
UNION ALL
   select id+1,
   a1-0.1*avg(2*x1*(a1*x1+a2*x2+a3*x3+a4*x4+a5*x5+a6*x6+b-y6)),
   a2-0.1*avg(2*x2*(a1*x1+a2*x2+a3*x3+a4*x4+a5*x5+a6*x6+b-y6)),
   a3-0.1*avg(2*x3*(a1*x1+a2*x2+a3*x3+a4*x4+a5*x5+a6*x6+b-y6)),
   a4-0.1*avg(2*x4*(a1*x1+a2*x2+a3*x3+a4*x4+a5*x5+a6*x6+b-y6)),
   a5-0.1*avg(2*x5*(a1*x1+a2*x2+a3*x3+a4*x4+a5*x5+a6*x6+b-y6)),
   a6-0.1*avg(2*x6*(a1*x1+a2*x2+a3*x3+a4*x4+a5*x5+a6*x6+b-y6)),
   b -0.1*avg(2*(a1*x1+a2*x2+a3*x3+a4*x4+a5*x5+a6*x6+b-y6))
   from gd, (select * from data) where id < 50 group by id,a1,a2,a3,a4,a5,a6,b)
select * from gd where id = 50 order by id;

\record gd.csv Automatically,6,100000,0.1,50
with recursive gd (id, a1, a2, a3, a4, a5, a6, b) as (
   select 0,1::float,1::float,1::float,1::float,1::float,1::float,1::float
UNION ALL
   select id+1,
   a1-0.1*avg(d_a1), a2-0.1*avg(d_a2), a3-0.1*avg(d_a3), a4-0.1*avg(d_a4), a5-0.1*avg(d_a5), a6-0.1*avg(d_a6), b-0.1*avg(d_b)
   from umbra.derivation(TABLE (
      select * from gd, (select * from data) where id < 50),
      lambda (x) ((x.a1 * x.x1 + x.a2 * x.x2 + x.a3 * x.x3 + x.a4 * x.x4 + x.a5 * x.x5 + x.a6 * x.x6 + x.b - x.y6)^2))
   group by id,a1,a2,a3,a4,a5,a6,b)
select * from gd where id = 50 order by id;

\record gd.csv Operator,6,100000,0.1,50
select * from umbra.gd(TABLE (select * from data), TABLE (select 1::float a1, 1::float a2, 1::float a3, 1::float a4, 1::float a5, 1::
      float a6, 1::float b), lambda (x,y) ((y.a1 * x.x1 + y.a2 * x.x2 + y.a3 * x.x3 + y.a4 * x.x4 + y.a5 * x.x5 + y.a6 * x.x6 + y.b -
      x.y6)^2), 50, 0.1, 0);
```

LISTING B.1: Gradient descent for linear regression.

```
create table if not exists data (x1 float, x2 float, y float);
insert into data (select *, round(1.0/(1+exp(-(-0.25+0.4*x1+0.2*x2))),0) from (select random() x1, random() x2 from generate_series
      (1,10)));

\record gd.csv Manually,2,10,0.5,150
with recursive gd (id, a1, a2, b) as (
   select 0,1::float,1::float,1::float
UNION ALL
   select id+1,
   a1-0.5*avg(2*x1*(sig(a1*x1+a2*x2+b)-y2)*(sig(a1*x1+a2*x2+b)*(1-sig(a1*x1+a2*x2+b)))),
   a2-0.5*avg(2*x2*(sig(a1*x1+a2*x2+b)-y2)*(sig(a1*x1+a2*x2+b)*(1-sig(a1*x1+a2*x2+b)))),
   b -0.5*avg(2*   (sig(a1*x1+a2*x2+b)-y2)*(sig(a1*x1+a2*x2+b)*(1-sig(a1*x1+a2*x2+b))))
   from gd, data where id < 150 group by id,a1,a2,b
),test as (select id, correct, count(*) from (select id, round(1.0/(1+exp(-a1*x1-a2*x2-b)),0)=y1 as correct from gd,data) group by id,
       correct)
select id, count*1.0/(select sum(count) from test t2 where t1.id=t2.id ) from test t1 where correct=true order by id;

\record gd.csv Automatically,2,10,0.5,150
with recursive gd (id, a1, a2, b) as (
   select 0,1::float,1::float,1::float
UNION ALL
   select id+1,
   a1-0.5*avg(d_a1), a2-0.5*avg(d_a2), b-0.5*avg(d_b)
   from umbra.derivation(TABLE (
      select * from gd, (select * from data) where id < 150),
      lambda (x) ((1.0/(1+exp(-(x.a1 * x.x1 +x.a2 * x.x2 + x.b))) - x.y)^2))
   group by id,a1,a2,b
),test as (select id, correct, count(*) from (select id, round(1.0/(1+exp(-a1*x1-a2*x2-b)),0)=y1 as correct from gd,data) group by id,
       correct)
select id, count*1.0/(select sum(count) from test t2 where t1.id=t2.id ) from test t1 where correct=true order by id;

\record gd.csv Operator,2,10,0.5,150
with gd as (
  select * from umbra.gd(TABLE (select * from data), TABLE (select 1::float a1, 1::float a2, 1::float b), lambda (x,y) ((1.0/(1+exp(-y
      .b - y.a1 * x.x1 - y.a2 * x.x2)) - x.y2)^2), 150, 0.5, 0)
),test as (select correct, count(*) from (select round(1.0/(1+exp(-a1*x1-a2*x2-b)),0)=y1 as correct from gd,data) group by correct)
select count*1.0/(select sum(count) from test t2) from test t1 where correct=true;
```

LISTING B.2: Gradient descent for logistic regression.

```
#!/bin/bash

# parameters
parallel=${PARALLEL:-"8"}
lr=0.001 # learningrate
maxatts=64
maxlimit=10000
attss="2 4 8 16 32 $maxatts"
iters="10 100 1000"
limits="10 100 1000 $maxlimit"
repeat=5

# create table for data
echo -n "create table if not exists data ("
for att in `seq 1 $maxatts`; do echo -n "x$att float, "; done
for att in `seq 1 $maxatts`; do echo -n "y$att float, "; done
echo "mynull float);"

# insert data
echo -n "insert into data (select *"
summe="0.5"
for att in `seq 1 $maxatts`; do summe="$summe+0.8*x${att}"; echo -n ",$summe"; done
echo -n " from (select "
for att in `seq 1 $maxatts`; do echo -n "random() x$att, "; done
echo -e "0 from generate_series(1,$maxlimit)));"

# benchmark loop
for limit in $limits; do
  for iter in $iters; do
    for atts in $attss; do

# build lambda
summe=""
for att in `seq 1 $atts`; do summe="$summe a${att}*x${att} +"; done
summe="$summe b-y$atts"
lambda=""
for att in `seq 1 $atts`; do lambda="$lambda x.a${att}*x.x${att} +"; done
lambda="$lambda x.b-x.y$atts"
lambda2=""
for att in `seq 1 $atts`; do lambda2="$lambda2 y.a${att}*x.x${att} +"; done
lambda2="$lambda2 y.b-x.y$atts"

# benchmark setting
echo \\set repeat $repeat
echo \\record gd.csv Manually,$atts,$limit,$lr,$iter,$parallel
# recursive table
echo -n "with recursive gd (id"
for att in `seq 1 $atts`; do echo -n ", a$att"; done
echo -n ", b) as (
   select 1, 1::float"
for att in `seq 1 $atts`; do echo -n ", 1::float"; done
# recursion step
echo "
UNION ALL
   select id+1,"
for att in `seq 1 $atts`; do echo "   a$att-$lr*avg(2*x$att*($summe)),"; done
echo -n "   b-$lr*avg(2*($summe))
   from gd, (select * from data limit $limit) where id < $iter group by id"
for att in `seq 1 $atts`; do echo -n ", a$att"; done
echo -n ", b)
select * from gd where id = $iter;
"

echo \\record gd.csv Automatically,$atts,$limit,$lr,$iter,$parallel
# recursive table
echo -n "with recursive gd (id"
for att in `seq 1 $atts`; do echo -n ", a$att"; done
echo -n ", b) as (
   select 1, 1::float"
for att in `seq 1 $atts`; do echo -n ", 1::float"; done
# recursion step
echo -n "
UNION ALL
   select id+1,"
for att in `seq 1 $atts`; do echo -n "a$att-$lr*avg(d_a$att),"; done
echo -n " b-$lr*avg(d_b)
   from umbra.derivation(TABLE (
      select * from gd, (select * from data limit $limit) where id < $iter),
      lambda (x) (($lambda)^2))
   group by id"
for att in `seq 1 $atts`; do echo -n ", a$att"; done
echo -n ", b)
select * from gd where id = $iter;
"

echo \\record gd.csv Operator,$atts,$limit,$lr,$iter,$parallel
echo -n "select * from umbra.labelling(TABLE (select * from data limit $limit), TABLE (select "
for att in `seq 1 $atts`; do echo -n "1::float a$att, "; done
echo "1::float b), lambda (x,y) (($lambda2)^2), $(($iter-1)), $lr, 0);"

    done
  done
done
```

LISTING B.3: Benchmarking linear regression (number of attributes).

```
#!/bin/bash

# parameters
parallel=${PARALLEL:-"8"}
lr=0.001 # learningrate
maxatts=64
maxlimit=10000
attss="2 4 8 16 32 $maxatts"
iters="10 100 1000"
limits="10 100 1000 $maxlimit"
repeat=5

# create table for data
echo -n "create table if not exists data ("
for att in `seq 1 $maxatts`; do echo -n "x$att float, "; done
for att in `seq 1 $maxatts`; do echo -n "y$att float, "; done
echo "mynull float);"

# insert data
echo -n "insert into data (select *"
summe="0.0"
for att in `seq 1 $maxatts`; do summe="$summe+x${att}"; echo -n ",round(($summe)/$att,0)"; done
echo -n " from (select "
for att in `seq 1 $maxatts`; do echo -n "random() x$att, "; done
echo -e "0 from generate_series(1,$maxlimit)));"

# benchmark loop
for limit in $limits; do
  for iter in $iters; do
    for atts in $attss; do

# build lambda
summe=""
for att in `seq 1 $atts`; do summe="$summe a${att}*x${att} +"; done
summe="2*(sig($summe b)-y$atts)*sig($summe b)*(1-sig($summe b))"
lambda=""
for att in `seq 1 $atts`; do lambda="$lambda x.a${att}*x.x${att} +"; done
lambda="sig($lambda x.b)-x.y$atts"
lambda2=""
for att in `seq 1 $atts`; do lambda2="$lambda2 y.a${att}*x.x${att} +"; done
lambda2="sig($lambda2 y.b)-x.y$atts"

# benchmark setting
echo \\set repeat $repeat
echo \\record gd_sigmoid.csv Manually,$atts,$limit,$lr,$iter,$parallel
# recursive table
echo -n "with recursive gd (id"
for att in `seq 1 $atts`; do echo -n ", a$att"; done
echo -n ", b) as (
   select 1, 1::float"
for att in `seq 1 $atts`; do echo -n ", 1::float"; done
# recursion step
echo "
UNION ALL
   select id+1,"
for att in `seq 1 $atts`; do echo "   a$att-$lr*avg(2*x$att*($summe)),"; done
echo -n "   b-$lr*avg(2*($summe))
   from gd, (select * from data limit $limit) where id < $iter group by id"
for att in `seq 1 $atts`; do echo -n ", a$att"; done
echo -n ", b)
select * from gd where id = $iter;
"

echo \\record gd_sigmoid.csv Automatically,$atts,$limit,$lr,$iter,$parallel
# recursive table
echo -n "with recursive gd (id"
for att in `seq 1 $atts`; do echo -n ", a$att"; done
echo -n ", b) as (
   select 1, 1::float"
for att in `seq 1 $atts`; do echo -n ", 1::float"; done
# recursion step
echo -n "
UNION ALL
   select id+1,"
for att in `seq 1 $atts`; do echo -n "a$att-$lr*avg(d_a$att),"; done
echo -n " b-$lr*avg(d_b)
   from umbra.derivation(TABLE (
      select * from gd, (select * from data limit $limit) where id < $iter),
      lambda (x) (($lambda)^2))
   group by id"
for att in `seq 1 $atts`; do echo -n ", a$att"; done
echo -n ", b)
select * from gd where id = $iter;
"

echo \\record gd_sigmoid.csv Operator,$atts,$limit,$lr,$iter,$parallel
echo -n "select * from umbra.labelling(TABLE (select * from data limit $limit), TABLE (select "
for att in `seq 1 $atts`; do echo -n "1::float a$att, "; done
echo "1::float b), lambda (x,y) (($lambda2)^2), $(($iter-1)), $lr, 0);"

    done
  done
done
```

LISTING B.4: Benchmarking logistic regression (number of attributes).

```
create table taxidata(id text, pickup_datetime date, dropoff_datetime text, passenger_count float, notdefined1 float, pickup_longitude
       float, pickup_latitude float, notdefined2 float, notdefined3 text, dropoff_longitude float, dropoff_latitude float, notdefined4
       float, notdefined5 float, notdefined6 float, notdefined7 float, notdefined8 float, store_and_fwd_flag float, trip_duration
      float);
\record taxibench.csv Load
copy taxidata from './taxidata.csv' delimiter ',';
\record taxibench.csv Extract
create view taxiprocessed as (select hour,day,trip_duration,ACOS(SIN(plat)*SIN(dlat)+COS(plat)*COS(dlat)*COS(dlong-plong))*6371000
      distance, ATAN2(SIN(dlong-plong)*COS(dlat),COS(plat)*SIN(dlat)-SIN(plat)*COS(dlat)*COS(dlong-plong))*180/PI() bearing from (
      select cast(substring(dropoff_datetime from 12 for 2) as float)  as hour,cast(substring(dropoff_datetime from 9 for 2) as float)
        as day,trip_duration, pickup_latitude/180*pi() plat, pickup_longitude/180*pi() plong, dropoff_latitude/180*pi() dlat,
      dropoff_longitude/180*pi() as dlong from taxidata));
\record taxibench.csv Normalise
create view taxinormalised(hour, day, distance, bearing, duration) as (
 select
  cast(hour as float)/(select max(hour)+1 from taxiprocessed),
  cast(day as float)/(select max(day) from taxiprocessed),
  distance/(select max(distance) from taxiprocessed where distance < 1000),
  (bearing+360)%360/360.0,
  trip_duration/(select max(trip_duration) from taxiprocessed)
 from taxiprocessed where distance < 1000
);
\record taxibench.csv Materialise
create table taxinormalised2( hour float, day float, distance float, bearing float, duration float);
insert into taxinormalised2 (select * from taxinormalised);

\record taxibench.csv Manually
with recursive gd (id, a1, a2, a3, a4, b) as (
   select 1, 1::float, 1::float, 1::float, 1::float, 1::float
UNION ALL
   select id+1,
   a1-0.001*avg(2*hour*( a1*hour + a2*day + a3*distance + a4*bearing + b-duration)),
   a2-0.001*avg(2*day*( a1*hour + a2*day + a3*distance + a4*bearing + b-duration)),
   a3-0.001*avg(2*distance*( a1*hour + a2*day + a3*distance + a4*bearing + b-duration)),
   a4-0.001*avg(2*bearing*( a1*hour + a2*day + a3*distance + a4*bearing + b-duration)),
   b -0.001*avg(2*( a1*hour + a2*day + a3*distance + a4*bearing + b-duration))
   from gd, (select * from taxinormalised2 tablesample reservoir (64)) where id < 37865 group by id, a1, a2, a3, a4, b)
select id, avg(a1*hour + a2*day + a3*distance + a4*bearing + b-duration)^2 from gd,taxinormalised2 where id = 37865 group by id order
      by id;

\record taxibench.csv Automatically
with recursive gd (id, a1, a2, a3, a4, b) as (
   select 1, 1::float, 1::float, 1::float, 1::float, 1::float
UNION ALL
   select id+1,a1-0.001*avg(d_a1),a2-0.001*avg(d_a2),a3-0.001*avg(d_a3),a4-0.001*avg(d_a4), b-0.001*avg(d_b)
   from umbra.derivation(TABLE (
      select * from gd, (select * from taxinormalised2 tablesample reservoir (64)) where id < 37865),
      lambda (x) (( x.a1*x.hour + x.a2*x.day + x.a3*x.distance + x.a4*x.bearing + x.b-x.duration)^2))
   group by id, a1, a2, a3, a4, b)
select id, avg(a1*hour + a2*day + a3*distance + a4*bearing + b-duration)^2 from gd,taxinormalised2 where id = 37865 group by id order
      by id;

\record taxibench.csv Operator
select avg(a1*hour + a2*day + a3*distance + a4*bearing + b-duration)^2
from taxinormalised2, umbra.gd(TABLE (select * from taxinormalised2), TABLE (select 1::float a1, 1::float a2, 1::float a3, 1::float a4
      , 1::float b), lambda (x,y) (( y.a1*x.hour + y.a2*x.day + y.a3*x.distance + y.a4*x.bearing + y.b-x.duration)^2), 37865, 0.001,
      64);
```

LISTING B.5: Linear regression on the New York taxi data.

```
create table if not exists iris (sepal_length float,sepal_width float,petal_length float,petal_width float,species int);
create table if not exists data (img float[], one_hot float[]);
copy iris from './iris.csv' delimiter ',' HEADER;
insert into data (select array[[sepal_length/10,sepal_width/10,petal_length/10,petal_width/10]] as img, array[(array_fill(0::float,
      array[species]) || 1::float ) || array_fill(0::float,array[2-species])] as one_hot from iris);

with recursive gd (id,w_xh,w_ho) as (
   select 0,(select array_agg(array_agg) from generate_series(1,4),(select array_agg(random()) from generate_series(1,20))),
            (select array_agg(array_agg) from generate_series(1,20),(select array_agg(random()) from generate_series(1,3)))
union all
   select id+1, w_xh - 0.2 * avg(transpose(img)*d_xh), w_ho - 0.2 * avg(transpose(a_xh)*d_ho)
   from ( select l_xh ** ( 1 - (a_xh ** a_xh)) as d_xh, *
          from ( select d_ho*transpose(w_ho) as l_xh, *
                 from ( select l_ho ** (1 - (a_ho ** a_ho)) as d_ho, *
                        from ( select 2 * (a_ho - one_hot) as l_ho, *
                               from ( select softmax(a_xh*w_ho) as a_ho, *
                                      from ( select tanh(img*w_xh) as a_xh, *
                                             from ( select * from data tablesample reservoir(50)),gd where id < 200))))))
   group by id, w_ho, w_xh
), test as (select id, correct, count(*) from (select id, highestposition(softmax(tanh(img*w_xh)*w_ho))=highestposition(one_hot) as
      correct from data,gd) group by id, correct)
select id, count*1.0/(select sum(count) from test t2 where t1.id=t2.id ) from test t1 where correct=true order by id;

with recursive gd (id,w_xh,w_ho) as (
   select 0,
      (select array_agg(array_agg) from generate_series(1,4),(select array_agg(random()) from generate_series(1,20))),
      (select array_agg(array_agg) from generate_series(1,20),(select array_agg(random()) from generate_series(1,3)))
union all
   select id+1, w_xh - 0.2 * avg(d_w_xh), w_ho - 0.2 * avg(d_w_ho)
   from  umbra.derivation(TABLE(select * from (select * from data tablesample reservoir(50)),gd where id < 200),lambda(x)(( softmax(
         tanh(x.img*x.w_xh)*x.w_ho) - one_hot)^2 ))
   group by id, w_ho, w_xh
), test as (select id, correct, count(*) from (select id, highestposition(softmax(tanh(img*w_xh)*w_ho))=highestposition(one_hot) as
      correct from data,gd) group by id, correct)
select id, count*1.0/(select sum(count) from test t2 where t1.id=t2.id ) from test t1 where correct=true order by id;
```

LISTING B.6: Neural network on Fisher's Iris flower data.

Bibliography

[1] Martín Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *CoRR* abs/1603.04467 (2016). arXiv: 1603.04467. URL: http://arxiv.org/abs/1603.04467.

[2] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. "Mind the Gap: Bridging Multi-Domain Query Workloads with EmptyHeaded". In: *Proc. VLDB Endow.* 10.12 (2017), pp. 1849–1852. URL: http://www.vldb.org/pvldb/vol10/p1849-aberger.pdf.

[3] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. "Mining Association Rules between Sets of Items in Large Databases". In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*. Ed. by Peter Buneman and Sushil Jajodia. ACM Press, 1993, pp. 207–216. DOI: 10.1145/170035.170072. URL: https://doi.org/10.1145/170035.170072.

[4] Alexa Internet. *wikipedia.org Traffic Statistics*. http://www.alexa.com/siteinfo/wikipedia.org. [Online; December 6, 2021]. 2021.

[5] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian J. Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. "Theano: new features and speed improvements". In: *CoRR* abs/1211.5590 (2012). arXiv: 1211.5590. URL: http://arxiv.org/abs/1211.5590.

[6] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. "The Multidimensional Database System RasDaMan". In: *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. 1998, pp. 575–577. DOI: 10.1145/276304.276386. URL: https://doi.org/10.1145/276304.276386.

[7] Atilim Gunes Baydin, Barak A. Pearlmutter, and Alexey Andreyevich Radul. "Automatic differentiation in machine learning: a survey". In: *CoRR* abs/1502.05767 (2015). arXiv: 1502.05767. URL: http://arxiv.org/abs/1502.05767.

[8] Rudolf Bayer and J. K. Metzger. "On the Encipherment of Search Trees and Random Access Files". In: *Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA*. 1975, p. 452. DOI: 10.1145/1282480.1282514. URL: http://doi.acm.org/10.1145/1282480.1282514.

[9] Anant P. Bhardwaj, Amol Deshpande, Aaron J. Elmore, David R. Karger, Sam Madden, Aditya G. Parameswaran, Harihar Subramanyam, Eugene Wu, and Rebecca Zhang. "Collaborative Data Analytics with DataHub". In: *Proc. VLDB Endow.* 8.12 (2015), pp. 1916–1919. URL: http://www.vldb.org/pvldb/vol8/p1916-bhardwaj.pdf.

[10] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya G. Parameswaran. "Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff". In: *Proc. VLDB Endow.* 8.12 (2015), pp. 1346–1357. URL: http://www.vldb.org/pvldb/vol8/p1346-bhattacherjee.pdf.

[11] Souvik Bhattacherjee and Amol Deshpande. "RStore: A Distributed Multi-Version Document Store". In: *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. 2018, pp. 389–400. DOI: 10.1109/ICDE.2018.00043. URL: https://doi.org/10.1109/ICDE.2018.00043.

[12] Carsten Binnig, Norman May, and Tobias Mindnich. "SQLScript: Efficiently Analyzing Big Enterprise Data in SAP HANA". 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*. Ed. by Volker Markl, Gunter Saake, Kai-Uwe Sattler, Gregor Hackenbroich, Bernhard Mitschang, Theo Härder, and Veit Köppen. Vol. P-214. LNI. GI, 2013, pp. 363–382. URL: https://dl.gi.de/20.500.12116/17332.

[13] Carsten Binnig, Robin Rehrmann, Franz Faerber, and Rudolf Riewe. "FunSQL: it is time to make SQL functional". In: *Proceedings of the 2012 Joint EDBT/ICDT Workshops, Berlin, Germany, March 30, 2012*. 2012, pp. 41–46. DOI: 10.1145/2320765.2320786. URL: https://doi.org/10.1145/2320765.2320786.

[14] Carsten Binnig, Abdallah Salama, Erfan Zamanian, Harald Kornmayer, Sven Listing, and Alexander C. Müller. "XDB - A Novel Database Architecture for Data Analytics as a Service". In: *2014 IEEE Big Data, Anchorage, AK, USA, June 27 - July 2, 2014*. 2014, pp. 96–103. DOI: 10.1109/BigData.Congress.2014.23. URL: https://doi.org/10.1109/BigData.Congress.2014.23.

[15] Altan Birler. "Scalable Reservoir Sampling on Many-Core CPUs". 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, 2019, pp. 1817–1819. DOI: 10.1145/3299869.3300096. URL: https://doi.org/10.1145/3299869.3300096.

[16] Mark Blacher, Joachim Giesen, Sören Laue, Julien Klaus, and Viktor Leis. "Machine Learning, Linear Algebra, and More: Is SQL All You Need?" In: *11th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022, Online Proceedings*. www.cidrdb.org, 2022.

[17] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. "SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle". In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf.

[18] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. "SystemML: Declarative Machine Learning on Spark". In: *Proc. VLDB Endow.* 9.13 (2016), pp. 1425–1436. DOI: 10.14778/3007263.3007279. URL: http://www.vldb.org/pvldb/vol9/p1425-boehm.pdf.

[19] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. "Scalable Garbage Collection for In-Memory MVCC Systems". In: *Proc. VLDB Endow.* 13.2 (2019), pp. 128–141. DOI: 10.14778/3364324.3364328. URL: http://www.vldb.org/pvldb/vol13/p128-bottcher.pdf.

[20] Sergey Brin and Lawrence Page. "The Anatomy of a Large-Scale Hypertextual Web Search Engine". In: *Computer Networks* 30.1-7 (1998), pp. 107–117. DOI: 10.1016/S0169-7552(98)00110-X. URL: https://doi.org/10.1016/S0169-7552(98)00110-X.

[21] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang Chiew Tan. "Archiving scientific data". In: *ACM Trans. Database Syst.* 29 (2004), pp. 2–42. DOI: 10.1145/974750.974752. URL: http://doi.acm.org/10.1145/974750.974752.

[22] Matthias Sebastian Bungeroth. "Conception of a Declarative Language for Machine Learning". Bachelor's Thesis. TUM, Oct. 2018.

[23] Dennis Butterstein and Torsten Grust. "Invest Once, Save a Million Times - LLVM-based Expression Compilation in PostgreSQL". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*. Ed. by Bernhard Mitschang, Daniela Nicklas, Frank Leymann, Harald Schöning, Melanie Herschel, Jens Teubner, Theo Härder, Oliver Kopp, and Matthias Wieland. Vol. P-265. LNI. GI, 2017, pp. 623–624. URL: https://dl.gi.de/20.500.12116/672.

[24] Dennis Butterstein and Torsten Grust. "Precision Performance Surgery for PostgreSQL: LLVM-based Expression Compilation, Just in Time". In: *Proc. VLDB Endow.* 9.13 (2016), pp. 1517–1520. DOI: 10.14778/3007263.3007298. URL: http://www.vldb.org/pvldb/vol9/p1517-butterstein.pdf.

[25] Amit Chavan, Silu Huang, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. "Towards a Unified Query Language for Provenance and Versioning". In: *7th USENIX Workshop on the Theory and Practice of Provenance, TaPP 2015, Edinburgh, Scotland, UK, July 8-9, 2015*. 2015. URL: https://www.usenix.org/conference/tapp15/workshop-program/presentation/chavan.

[26] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems". In: *CoRR* abs/1512.01274 (2015). arXiv: 1512.01274. URL: http://arxiv.org/abs/1512.01274.

[27] Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. "HetExchange: Encapsulating heterogeneous CPU-GPU parallelism in JIT compiled engines". In: *Proc. VLDB Endow.* 12.5 (2019), pp. 544–556. DOI: 10.14778/3303753.3303760. URL: http://www.vldb.org/pvldb/vol12/p544-chrysogelos.pdf.

[28] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Christopher Ré, and Matei Zaharia. "Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark". In: *ACM SIGOPS Oper. Syst. Rev.* 53.1 (2019), pp. 14–25. DOI: 10.1145/3352020.3352024. URL: https://doi.org/10.1145/3352020.3352024.

[29] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Çetintemel, and Stan Zdonik. "An Architecture for Compiling UDF-centric Workflows". In: *Proc. VLDB Endow.* 8.12 (2015), pp. 1466–1477. URL: http://www.vldb.org/pvldb/vol8/p1466-crotty.pdf.

[30] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. "Tupleware: "Big" Data, Big Analytics, Small Clusters". In: *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. 2015. URL: http://cidrdb.org/cidr2015/Papers/CIDR15_Paper23u.pdf.

[31] Andrew Crotty, Alex Galakatos, and Tim Kraska. "Tupleware: Distributed Machine Learning on Small Clusters". In: *IEEE Data Eng. Bull.* 37.3 (2014), pp. 63–76. URL: http://sites.computer.org/debull/A14sept/p63.pdf.

[32] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. "Vizdom: Interactive Analytics through Pen and Touch". In: *Proc. VLDB Endow.* 8.12 (2015), pp. 2024–2027. DOI: 10.14778/2824032.2824127. URL: http://www.vldb.org/pvldb/vol8/p2024-crotty.pdf.

[33] Philippe Cudre-Mauroux, Hideaki Kimura, Kian-Tat Lim, Jennie Rogers, Samuel Madden, Michael Stonebraker, Stanley B Zdonik, and Paul G Brown. "SS-DB: A standard science dbms benchmark". In: *Under submission* (2010).

[34] Philippe Cudré-Mauroux, Hideaki Kimura, Kian-Tat Lim, Jennie Rogers, Roman Simakov, Emad Soroush, Pavel Velikhov, Daniel L. Wang, Magdalena Balazinska, Jacek Becla, David J. DeWitt, Bobbi Heath, David Maier, Samuel Madden, Jignesh M. Patel, Michael Stonebraker, and Stanley B. Zdonik. "A Demonstration of SciDB: A Science-Oriented DBMS". In: *Proc. VLDB Endow.* 2.2 (2009), pp. 1534–1537. DOI: 10.14778/1687553.1687584. URL: http://www.vldb.org/pvldb/vol2/vldb09-76.pdf.

[35] Leandro F. Cupertino, Cleomar Pereira da Silva, Douglas Mota Dias, Marco Aurélio Cavalcanti Pacheco, and Cristiana Bentes. "Evolving CUDA PTX programs by quantum inspired linear genetic programming". In: *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Companion Material Proceedings, Dublin, Ireland, July 12-16, 2011*. Ed. by Natalio Krasnogor and Pier Luca Lanzi. ACM, 2011, pp. 399–406. DOI: 10.1145/2001858.2002026. URL: https://doi.org/10.1145/2001858.2002026.

[36] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Tilmann Rabl, and Volker Markl. "Continuous Deployment of Machine Learning Pipelines". In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. 2019, pp. 397–408. DOI: 10.5441/002/edbt.2019.35. URL: https://doi.org/10.5441/002/edbt.2019.35.

[37] Oksana Dolmatova, Nikolaus Augsten, and Michael H. Böhlen. "A Relational Matrix Algebra and its Implementation in a Column Store". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 2573–2587. DOI: 10.1145/3318464.3389747. URL: https://doi.org/10.1145/3318464.3389747.

[38] Jennie Duggan, Olga Papaemmanouil, Leilani Battle, and Michael Stonebraker. "Skew-Aware Join Optimization for Array Databases". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 123–135. DOI: 10.1145/2723372.2723709. URL: https://doi.org/10.1145/2723372.2723709.

[39] Christian Duta and Torsten Grust. "Functional-Style SQL UDFs With a Capital 'F'". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 1273–1287. DOI: 10.1145/3318464.3389707. URL: https://doi.org/10.1145/3318464.3389707.

[40] Christian Duta, Denis Hirn, and Torsten Grust. "Compiling PL/SQL Away". In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: http://cidrdb.org/cidr2020/papers/p1-duta-cidr20.pdf.

[41] Andrew Eisenberg, Jim Melton, Krishna G. Kulkarni, Jan-Eike Michels, and Fred Zemke. "SQL: 2003 has been published". In: *SIGMOD Rec.* 33.1 (2004), pp. 119–126. DOI: 10.1145/974121.974142. URL: https://doi.org/10.1145/974121.974142.

[42] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. "Towards a unified architecture for in-RDBMS analytics". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. Ed. by K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman. ACM, 2012, pp. 325–336. DOI: 10.1145/2213836.2213874. URL: https://doi.org/10.1145/2213836.2213874.

[43] Ronald A Fisher. "The use of multiple measurements in taxonomic problems". In: *Annals of eugenics* 7.2 (1936), pp. 179–188.

[44] Zekai J. Gao, Shangyu Luo, Luis Leopoldo Perez, and Chris Jermaine. "The BUDS Language for Distributed Bayesian Machine Learning". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. Ed. by Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu. ACM, 2017, pp. 961–976. DOI: 10.1145/3035918.3035937. URL: https://doi.org/10.1145/3035918.3035937.

[45] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. "Large-scale matrix factorization with distributed stochastic gradient descent". In: *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*. Ed. by Chid Apté, Joydeep Ghosh, and Padhraic Smyth. ACM, 2011, pp. 69–77. DOI: 10.1145/2020408.2020426. URL: https://doi.org/10.1145/2020408.2020426.

[46] George Giorgidze, Torsten Grust, Alexander Ulrich, and Jeroen Weijers. "Algebraic data types for language-integrated queries". In: *Proceedings of the 2013 Workshop on Data Driven Functional Programming, DDFP 2013, Rome, Italy, January 22, 2013*. Ed. by Evelyne Viegas, Karin K. Breitman, and Judith Bishop. ACM, 2013, pp. 5–10. DOI: 10.1145/2429376.2429379. URL: https://doi.org/10.1145/2429376.2429379.

[47] Goetz Graefe. "Encapsulation of Parallelism in the Volcano Query Processing System". In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990*. Ed. by Hector Garcia-Molina and H. V. Jagadish. ACM Press, 1990, pp. 102–111. DOI: 10.1145/93597.98720. URL: https://doi.org/10.1145/93597.98720.

[48] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. "FERRY: database-supported program execution". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*. Ed. by Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul. ACM, 2009, pp. 1063–1066. DOI: 10.1145/1559845.1559982. URL: https://doi.org/10.1145/1559845.1559982.

[49] Torsten Grust, Nils Schweinsberg, and Alexander Ulrich. "Functions Are Data Too (Defunctionalization for PL/SQL)". In: *Proc. VLDB Endow.* 6.12 (2013), pp. 1214–1217. DOI: 10.14778/2536274.2536279. URL: http://www.vldb.org/pvldb/vol6/p1214-grust.pdf.

[50] Torsten Grust and Alexander Ulrich. "First-Class Functions for First-Order Database Engines". In: *Proceedings of the 14th International Symposium on Database Programming Languages (DBPL 2013), August 30, 2013, Riva del Garda, Trento, Italy*. Ed. by Todd J. Green and Alan Schmitt. 2013. URL: http://arxiv.org/abs/1308.0158.

[51] Tobias Götz. "Benchmarking Array Database Systems". Guided Research. TUM, Sept. 2020.

[52] Tobias Götz. "Integration of ArrayQL in a Main-Memory Database System". Bachelor's Thesis. TUM, Mar. 2020.

[53] Donghyoung Han, Yoon-Min Nam, Jihye Lee, Kyongseok Park, Hyunwoo Kim, and Min-Soo Kim. "DistME: A Fast and Elastic Distributed Matrix Computation Engine using GPUs". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. 2019, pp. 759–774. DOI: 10.1145/3299869.3319865. URL: https://doi.org/10.1145/3299869.3319865.

[54] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. "The MADlib Analytics Library or MAD Skills, the SQL". In: *Proc. VLDB Endow.* 5.12 (2012), pp. 1700–1711. URL: http://vldb.org/pvldb/vol5/p1700_joehellerstein_vldb2012.pdf.

[55] Thomas Heyenbrock. "Efficient statistical methods for database systems". Study Thesis. TUM, 2018.

[56] Denis Hirn and Torsten Grust. "One WITH RECURSIVE is Worth Many GOTOs". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2021, pp. 723–735. DOI: 10.1145/3448016.3457272. URL: https://doi.org/10.1145/3448016.3457272.

[57] Denis Hirn and Torsten Grust. "PgCuckoo: Laying Plan Eggs in PostgreSQL's Nest". 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, 2019, pp. 1929–1932. DOI: 10.1145/3299869.3320211. URL: https://doi.org/10.1145/3299869.3320211.

[58] Denis Hirn and Torsten Grust. "PL/SQL Without the PL". In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 2677–2680. DOI: 10.1145/3318464.3384678. URL: https://doi.org/10.1145/3318464.3384678.

[59] Silu Huang, Chi Wang, Bolin Ding, and Surajit Chaudhuri. "Efficient Identification of Approximate Best Configuration of Training in Large Datasets". In: *The Thirty-Third AAAI Conference on Artificial Intelligence, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. 2019, pp. 3862–3869. URL: https://aaai.org/ojs/index.php/AAAI/article/view/4274.

[60] Silu Huang, Liqi Xu, Jialin Liu, Aaron J. Elmore, and Aditya G. Parameswaran. "OrpheusDB: Bolt-on Versioning for Relational Databases". In: *Proc. VLDB Endow.* 10.10 (2017), pp. 1130–1141. URL: http://www.vldb.org/pvldb/vol10/p1130-huang.pdf.

[61] Nina Hubig, Linnea Passing, Maximilian E. Schüle, Dimitri Vorona, Alfons Kemper, and Thomas Neumann. "HyPerInsight: Data Exploration Deep Inside HyPer". In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management, CIKM 2017, Singapore, November 06 - 10, 2017*. Ed. by Ee-Peng Lim, Marianne Winslett, Mark Sanderson, Ada Wai-Chee Fu, Jimeng Sun, J. Shane Culpepper, Eric Lo, Joyce C. Ho, Debora Donato, Rakesh Agrawal, Yu Zheng, Carlos Castillo, Aixin Sun, Vincent S. Tseng, and Chenliang Li. ACM, 2017, pp. 2467–2470. DOI: 10.1145/3132847.3133167. URL: https://doi.org/10.1145/3132847.3133167.

[62] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. "Declarative Recursive Computation on an RDBMS, or, Why You Should Use a Database For Distributed Machine Learning". In: *CoRR* abs/1904.11121 (2019).

[63] Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, Saravanan Thirumuruganathan, Sanjay Chawla, and Divy Agrawal. "A Cost-based Optimizer for Gradient Descent Optimization". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 2017, pp. 977–992. DOI: 10.1145/3035918.3064042. URL: http://doi.acm.org/10.1145/3035918.3064042.

[64] Konstantinos Karanasos, Matteo Interlandi, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Doris Xin, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. "Extending Relational Query Processing with ML Inference". In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: http://cidrdb.org/cidr2020/papers/p24-karanasos-cidr20.pdf.

[65] Lukas Karnowski. "High-performance Version Controlling for Database Systems". Bachelor's Thesis. TUM, Oct. 2017.

[66] Lukas Karnowski. "Improving Version Control in Database Systems". Guided Research. TUM, Sept. 2019.

[67] Lukas Karnowski, Maximilian E. Schüle, Alfons Kemper, and Thomas Neumann. "Umbra as a Time Machine". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2021), 19. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 13.-17. September 2021, Dresden, Germany, Proceedings*. Ed. by Kai-Uwe Sattler, Melanie Herschel, and Wolfgang Lehner. Vol. P-311. LNI. Gesellschaft für Informatik, Bonn, 2021, pp. 123–132. DOI: 10.18420/btw2021-06. URL: https://doi.org/10.18420/btw2021-06.

[68] Alfons Kemper. *Datenbanksysteme - Eine Einführung, 10. Auflage*. De Gruyter Studium. de Gruyter Oldenbourg, 2015. ISBN: 978-3-11-044375-2. URL: https://www.degruyter.com/document/isbn/9783110443752/html.

[69] 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*. Ed. by Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan. IEEE Computer Society, 2011, pp. 195–206. DOI: 10.1109/ICDE.2011.5767867. URL: https://doi.org/10.1109/ICDE.2011.5767867.

[70] Alfons Kemper, Thomas Neumann, Jan Finis, Florian Funke, Viktor Leis, Henrik Mühe, Tobias Mühlbauer, and Wolf Rödiger. "Processing in the Hybrid OLTP & OLAP Main-Memory Database System HyPer". In: *IEEE Data Eng. Bull.* 36.2 (2013), pp. 41–47. URL: http://sites.computer.org/debull/A13june/hyper1.pdf.

[71] David Kernert, Frank Köhler, and Wolfgang Lehner. "Bringing Linear Algebra Objects to Life in a Column-Oriented In-Memory Database". In: *Proceedings of the 1st International Workshop on In Memory Data Management and Analytics, IMDM 2013, Riva Del Garda, Italy, August 26, 2013*. 2013, pp. 37–49. URL: http://www-db.in.tum.de/other/imdm2013/papers/Kernert.pdf.

[72] David Kernert, Frank Köhler, and Wolfgang Lehner. "SLACID - sparse linear algebra in a column-oriented in-memory database system". In: *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*. 2014, 11:1–11:12. DOI: 10.1145/2618243.2618254. URL: http://doi.acm.org/10.1145/2618243.2618254.

[73] Mijung Kim and K. Selçuk Candan. "TensorDB: In-Database Tensor Manipulation with Tensor-Relational Query Plans". In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*. 2014, pp. 2039–2041. DOI: 10.1145/2661829.2661842. URL: http://doi.acm.org/10.1145/2661829.2661842.

[74] Benedikt Kleiner. "Versioning for Databases". Seminar: Techniques for implementing main memory database systems. Seminar's Thesis. TUM, Dec. 2017. URL: https://db.in.tum.de/teaching/ws1718/seminarHauptspeicherdbs/paper/kleiner.pdf.

[75] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter R. Pietzuch. "Crossbow: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers". In: *Proc. VLDB Endow.* 12.11 (2019), pp. 1399–1413. DOI: 10.14778/3342263.3342276. URL: http://www.vldb.org/pvldb/vol12/p1399-koliousis.pdf.

[76] Krishna G. Kulkarni and Jan-Eike Michels. "Temporal features in SQL: 2011". In: *SIGMOD Record* 41.3 (2012), pp. 34–43. DOI: 10.1145/2380776.2380786. URL: http://doi.acm.org/10.1145/2380776.2380786.

[77] Andreas Kunft, Alexander Alexandrov, Asterios Katsifodimos, and Volker Markl. "Bridging the gap: towards optimization across linear and relational algebra". In: *Proceedings of the 3rd ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond, BeyondMR@SIGMOD 2016, San Francisco, CA, USA, July 1, 2016*. Ed. by Foto N. Afrati, Jacek Sroka, and Jan Hidders. ACM, 2016, p. 1. DOI: 10.1145/2926534.2926540. URL: https://doi.org/10.1145/2926534.2926540.

[78] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. "An Intermediate Representation for Optimizing Machine Learning Pipelines". In: *Proc. VLDB Endow.* 12.11 (2019), pp. 1553–1567. DOI: 10.14778/3342263.3342633. URL: http://www.vldb.org/pvldb/vol12/p1553-kunft.pdf.

[79] Sören Laue, Matthias Mitterreiter, and Joachim Giesen. "Computing Higher Order Derivatives of Matrix and Tensor Expressions". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. 2018, pp. 2755–2764. URL: https://proceedings.neurips.cc/paper/2018/hash/0a1bf96b7165e962e90cb14648c9462d-Abstract.html.

[80] Matthew Lease, James Allan, and W. Bruce Croft. "Regression Rank: Learning to Meet the Opportunity of Descriptive Queries". In: *Advances in Information Retrieval, 31th European Conference on IR Research, ECIR 2009, Toulouse, France, April 6-9, 2009. Proceedings*. Ed. by Mohand Boughanem, Catherine Berrut, Josiane Mothe, and Chantal Soulé-Dupuy. Vol. 5478. Lecture Notes in Computer Science. Springer, 2009, pp. 90–101. DOI: 10.1007/978-3-642-00958-7_11. URL: https://doi.org/10.1007/978-3-642-00958-7_11.

[81] 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*. 2013, pp. 38–49. DOI: 10.1109/ICDE.2013.6544812. URL: https://doi.org/10.1109/ICDE.2013.6544812.

[82] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. "Efficient Processing of Window Functions in Analytical SQL Queries". In: *Proc. VLDB Endow.* 8.10 (2015), pp. 1058–1069. DOI: 10.14778/2794367.2794375. URL: http://www.vldb.org/pvldb/vol8/p1058-leis.pdf.

[83] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. "Cardinality Estimation Done Right: Index-Based Join Sampling". In: *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017. URL: http://cidrdb.org/cidr2017/papers/p9-leis-cidr17.pdf.

[84] Tony E. Lewis and George D. Magoulas. "TMBL kernels for CUDA GPUs compile faster using PTX: computational intelligence on consumer games and graphics hardware". In: *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Companion Material Proceedings, Dublin, Ireland, July 12-16, 2011*. Ed. by Natalio Krasnogor and Pier Luca Lanzi. ACM, 2011, pp. 455–462. DOI: 10.1145/2001858.2002033. URL: https://doi.org/10.1145/2001858.2002033.

[85] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. "MLog: Towards Declarative In-Database Machine Learning". In: *Proc. VLDB Endow.* 10.12 (2017), pp. 1933–1936. URL: http://www.vldb.org/pvldb/vol10/p1933-zhang.pdf.

[86] Tyng-Yeu Liang, Hung-Fu Li, and Bi-Shing Chen. "A Distributed PTX Compilation and Execution System on Hybrid CPU/GPU Clusters". In: *Intelligent Systems and Applications - Proceedings of the International Computer Symposium (ICS) held at Taichung, Taiwan, December 12-14, 2014*. Ed. by William Cheng-Chung Chu, Han-Chieh Chao, and Stephen Jenn-Hwa Yang. Vol. 274. Frontiers in Artificial Intelligence and Applications. IOS Press, 2014, pp. 1355–1364. DOI: 10.3233/978-1-61499-484-8-1355. URL: https://doi.org/10.3233/978-1-61499-484-8-1355.

[87] Kian-Tat Lim, David Maier, J. Becla, Martin Kersten, Y. Zhang, and Michael Stonebraker. "ArrayQL syntax". In: *XLDB*. 2012. URL: http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL-Draft-4.pdf.

[88] Qian Lin, Kaiyuan Yang, Tien Tuan Anh Dinh, Qingchao Cai, Gang Chen, Beng Chin Ooi, Pingcheng Ruan, Sheng Wang, Zhongle Xie, Meihui Zhang, and Olafs Vandans. "ForkBase: Immutable, Tamper-evident Storage Substrate for Branchable Applications". In: *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 1718–1721. DOI: 10.1109/ICDE48307.2020.00153. URL: https://doi.org/10.1109/ICDE48307.2020.00153.

[89] David B. Lomet, Roger S. Barga, Mohamed F. Mokbel, German Shegalov, Rui Wang, and Yunyue Zhu. "Transaction Time Support Inside a Database Engine". In: *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*. 2006, p. 35. DOI: 10.1109/ICDE.2006.162. URL: https://doi.org/10.1109/ICDE.2006.162.

[90] Kevin Loney. *Oracle Database 11g The Complete Reference*. McGraw-Hill, Inc., 2008.

[91] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, Luis Leopoldo Perez, and Christopher M. Jermaine. "Scalable Linear Algebra on a Relational Database System". In: *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. 2017, pp. 523–534. DOI: 10.1109/ICDE.2017.108. URL: https://doi.org/10.1109/ICDE.2017.108.

[92] Daniel Lustig, Sameer Sahasrabuddhe, and Olivier Giroux. "A Formal Analysis of the NVIDIA PTX Memory Consistency Model". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. Ed. by Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck. ACM, 2019, pp. 257–270. DOI: 10.1145/3297858.3304043. URL: https://doi.org/10.1145/3297858.3304043.

[93] Michael Maddox, David Goehring, Aaron J. Elmore, Samuel Madden, Aditya G. Parameswaran, and Amol Deshpande. "Decibel: The Relational Dataset Branching System". In: *Proc. VLDB Endow.* 9.9 (2016), pp. 624–635. URL: http://www.vldb.org/pvldb/vol9/p624-maddox.pdf.

[94] David Maier, Peter Baumann, Martin Kersten, Kian-Tat Lim, and Michael Stonebraker. "ArrayQL algebra: version 3". In: *XLDB*. 2012. URL: http://www.xldb.org/wp-content/uploads/2012/09/ArrayQL_Algebra_v3+.pdf.

[95] Nantia Makrynioti, Nikolaos Vasiloglou, Emir Pasalic, and Vasilis Vassalos. "Modelling Machine Learning Algorithms on Relational Data with Datalog". In: *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM @SIGMOD 2018, Houston, TX, USA, June 15, 2018*. Ed. by Sebastian Schelter, Stephan Seufert, and Arun Kumar. ACM, 2018, 5:1–5:4. DOI: 10.1145/3209889.3209893. URL: https://doi.org/10.1145/3209889.3209893.

[96] Norman May, Alexander Böhm, and Wolfgang Lehner. "SAP HANA - The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings*. 2017, pp. 545–563. URL: https://dl.gi.de/20.500.12116/656.

[97] Jan-Eike Michels, Keith Hare, Krishna G. Kulkarni, Calisto Zuzarte, Zhen Hua Liu, Beda Christoph Hammerschmidt, and Fred Zemke. "The New and Improved SQL: 2016 Standard". In: *SIGMOD Rec.* 47.2 (2018), pp. 51–60. DOI: 10.1145/3299887.3299897. URL: https://doi.org/10.1145/3299887.3299897.

[98] Iain Murray. *Machine Learning and Pattern Recognition (MLPR): Backpropagation of Derivatives*. 2017. URL: https://www.inf.ed.ac.uk/teaching/courses/mlpr/2017/notes/w5a_backprop.pdf (visited on 09/02/2021).

[99] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware". In: *Proc. VLDB Endow.* 4.9 (2011), pp. 539–550. DOI: 10.14778/2002938.2002940. URL: http://www.vldb.org/pvldb/vol4/p539-neumann.pdf.

[100] Thomas Neumann and Michael J. Freitag. "Umbra: A Disk-Based System with In-Memory Performance". In: *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020. URL: http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf.

[101] Terence John Parr and Russell W. Quong. "ANTLR: A Predicated- *LL(k)* Parser Generator". In: *Softw. Pract. Exp.* 25.7 (1995), pp. 789–810. DOI: 10.1002/spe.4380250705. URL: https://doi.org/10.1002/spe.4380250705.

[102] Linnea Passing, Manuel Then, Nina Hubig, Harald Lang, Michael Schreier, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. "SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases". In: *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*. Ed. by Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß. OpenProceedings.org, 2017, pp. 84–95. DOI: 10.5441/002/edbt.2017.09. URL: https://doi.org/10.5441/002/edbt.2017.09.

[103] Holger Pirk, Ying Zhang, Stefan Manegold, and Martin Kersten. *An evaluation of ad-hoc queries on arrays in MonetDB*. 2013.

[104] Magdalena Pröbstl, Philipp Fent, Maximilian E. Schüle, Moritz Sichert, Thomas Neumann, and Alfons Kemper. "One Buffer Manager to Rule Them All: Using Distributed Memory with Cache Coherence over RDMA". In: *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2021, Copenhagen, Denmark, August 16, 2021*. Ed. by Rajesh Bordawekar and Tirthankar Lahiri. 2021, pp. 17–26. URL: http://www.adms-conf.org/2021-camera-ready/probstl_adms21.pdf.

[105] Benjamin Recht, Christopher Ré, Stephen J. Wright, and Feng Niu. "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". In: *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain*. Ed. by John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando C. N. Pereira, and Kilian Q. Weinberger. 2011, pp. 693–701. URL: https://proceedings.neurips.cc/paper/2011/hash/218a0aefd1d1a4be65601cc6ddc1520e-Abstract.html.

[106] Marc J. Rochkind. "The Source Code Control System". In: *IEEE Trans. Software Eng.* 1.4 (1975), pp. 364–370. DOI: 10.1109/TSE.1975.6312866. URL: https://doi.org/10.1109/TSE.1975.6312866.

[107] Betty Salzberg and Vassilis J. Tsotras. "Comparison of Access Methods for Time-Evolving Data". In: *ACM Comput. Surv.* 31.2 (1999), pp. 158–221. DOI: 10.1145/319806.319816. URL: http://doi.acm.org/10.1145/319806.319816.

[108] Maximilian Schleich, Dan Olteanu, Mahmoud Abo Khamis, Hung Q. Ngo, and Xuan-Long Nguyen. "A Layered Aggregate Engine for Analytics Workloads". 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, 2019, pp. 1642–1659. DOI: 10.1145/3299869.3324961. URL: https://doi.org/10.1145/3299869.3324961.

[109] Josef Schmeißer, Maximilian E. Schüle, Viktor Leis, Thomas Neumann, and Alfons Kemper. "B^2-Tree: Cache-Friendly String Indexing within B-Trees". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2021), 19. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 13.-17. September 2021, Dresden, Germany, Proceedings*. Ed. by Kai-Uwe Sattler, Melanie Herschel, and Wolfgang Lehner. Vol. P-311. LNI. Gesellschaft für Informatik, Bonn, 2021, pp. 39–58. DOI: 10.18420/btw2021-02. URL: https://doi.org/10.18420/btw2021-02.

[110] Josef Schmeißer. "Integration of a Version Control Inside of Modern Database Systems". Guided Research. TUM, Mar. 2019.

[111] Tom Schreiber, Simone Bonetti, Torsten Grust, Manuel Mayr, and Jan Rittinger. "Thirteen New Players in the Team: A Ferry-based LINQ to SQL Provider". In: *Proc. VLDB Endow.* 3.2 (2010), pp. 1549–1552. DOI: 10.14778/1920841.1921035. URL: http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/D09.pdf.

[112] Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN". In: *ACM Trans. Database Syst.* 42.3 (2017). DOI: 10.1145/3068335. URL: http://doi.acm.org/10.1145/3068335.

[113] Maximilian Schüle, Pascal Schliski, Thomas Hutzelmann, Tobias Rosenberger, Viktor Leis, Dimitri Vorona, Alfons Kemper, and Thomas Neumann. "Monopedia: Staying Single is Good Enough - The HyPer Way for Web Scale Applications". In: *Proc. VLDB Endow.* 10.12 (2017), pp. 1921–1924. DOI: 10.14778/3137765.3137809. URL: http://www.vldb.org/pvldb/vol10/p1921-schuele.pdf.

[114] Maximilian E. Schüle, Matthias Bungeroth, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. "MLearn: A Declarative Machine Learning Language for Database Systems". In: *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning, DEEM@SIGMOD 2019, Amsterdam, The Netherlands, June 30, 2019*. Ed. by Sebastian Schelter, Neoklis Polyzotis, Stephan Seufert, and Manasi Vartak. ACM, 2019, 7:1–7:4. DOI: 10.1145/3329486.3329494. URL: https://doi.org/10.1145/3329486.3329494.

[115] Maximilian E. Schüle, Matthias Bungeroth, Dimitri Vorona, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. "ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python". In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. Ed. by Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi. OpenProceedings.org, 2019, pp. 562–565. DOI: 10.5441/002/edbt.2019.56. URL: https://doi.org/10.5441/002/edbt.2019.56.

[116] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. "ArrayQL for Linear Algebra within Umbra". In: *SSDBM 2021: 33rd International Conference on Scientific and Statistical Database Management, Tampa, FL, USA, July 6-7, 2021*. Ed. by Qiang Zhu, Xingquan Zhu, Yicheng Tu, Zichen Xu, and Anand Kumar. ACM, 2021, pp. 193–196. DOI: 10.1145/3468791.3468838. URL: https://doi.org/10.1145/3468791.3468838.

[117] Maximilian E. Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. "ArrayQL Integration into Code-Generating Database Systems". In: *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*. Ed. by Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang. OpenProceedings.org, 2022, 1:40–1:51. DOI: 10.5441/002/edbt.2022.04. URL: https://doi.org/10.5441/002/edbt.2022.04.

[118] Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. "Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL". In: *SSDBM 2020: 32nd International Conference on Scientific and Statistical Database Management, Vienna, Austria, July 7-9, 2020*. Ed. by Elaheh Pourabbas, Dimitris Sacharidis, Kurt Stockinger, and Thanasis Vergoulis. ACM, 2020, 6:1–6:12. DOI: 10.1145/3400903.3400915. URL: https://doi.org/10.1145/3400903.3400915.

[119] Maximilian E. Schüle, Lukas Karnowski, Josef Schmeißer, Benedikt Kleiner, Alfons Kemper, and Thomas Neumann. "Versioning in Main-Memory Database Systems: From MusaeusDB to TardisDB". In: *Proceedings of the 31st International Conference on Scientific and Statistical Database Management, SSDBM 2019, Santa Cruz, CA, USA, July 23-25, 2019*. Ed. by Carlos Maltzahn and Tanu Malik. ACM, 2019, pp. 169–180. DOI: 10.1145/3335783.3335792. URL: https://doi.org/10.1145/3335783.3335792.

[120] Maximilian E. Schüle, Alex Kulikov, Alfons Kemper, and Thomas Neumann. "ARTful Skyline Computation for In-Memory Database Systems". In: *New Trends in Databases and Information Systems - ADBIS 2020 Short Papers, Lyon, France, August 25-27, 2020, Proceedings*. Ed. by Jérôme Darmont, Boris Novikov, and Robert Wrembel. Vol. 1259. Communications in Computer and Information Science. Springer, 2020, pp. 3–12. DOI: 10.1007/978-3-030-54623-6_1. URL: https://doi.org/10.1007/978-3-030-54623-6_1.

[121] Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. "In-Database Machine Learning with SQL on GPUs". In: *SSDBM 2021: 33rd International Conference on Scientific and Statistical Database Management, Tampa, FL, USA, July 6-7, 2021*. Ed. by Qiang Zhu, Xingquan Zhu, Yicheng Tu, Zichen Xu, and Anand Kumar. ACM, 2021, pp. 25–36. DOI: 10.1145/3468791.3468840. URL: https://doi.org/10.1145/3468791.3468840.

[122] Maximilian E. Schüle, Linnea Passing, Alfons Kemper, and Thomas Neumann. "Ja-(zu-)SQL: Evaluation einer SQL-Skriptsprache für Hauptspeicherdatenbanksysteme". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings*. Ed. by Torsten Grust, Felix Naumann, Alexander Böhm, Wolfgang Lehner, Theo Härder, Erhard Rahm, Andreas Heuer, Meike Klettke, and Holger Meyer. Vol. P-289. LNI. Gesellschaft für Informatik, Bonn, 2019, pp. 107–126. DOI: 10.18420/btw2019-08. URL: https://doi.org/10.18420/btw2019-08.

[123] Maximilian E. Schüle, Josef Schmeißer, Thomas Blum, Alfons Kemper, and Thomas Neumann. "TardisDB: Extending SQL to Support Versioning". In: *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. Ed. by Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava. ACM, 2021, pp. 2775–2778. DOI: 10.1145/3448016.3452767. URL: https://doi.org/10.1145/3448016.3452767.

[124] Maximilian E. Schüle, Frédéric Simonis, Thomas Heyenbrock, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. "In-Database Machine Learning: Gradient Descent and Tensor Algebra for Main Memory Database Systems". In: *Datenbanksysteme für Business, Technologie und Web (BTW 2019), 18. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 4.-8. März 2019, Rostock, Germany, Proceedings*. Ed. by Torsten Grust, Felix Naumann, Alexander Böhm, Wolfgang Lehner, Theo Härder, Erhard Rahm, Andreas Heuer, Meike Klettke, and Holger Meyer. Vol. P-289. LNI. Gesellschaft für Informatik, Bonn, 2019, pp. 247–266. DOI: 10.18420/btw2019-16. URL: https://doi.org/10.18420/btw2019-16.

[125] Maximilian E. Schüle, Dimitri Vorona, Linnea Passing, Harald Lang, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. "The Power of SQL Lambda Functions". In: *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. Ed. by Melanie Herschel, Helena Galhardas, Berthold Reinwald, Irini Fundulaki, Carsten Binnig, and Zoi Kaoudi. OpenProceedings.org, 2019, pp. 534–537. DOI: 10.5441/002/edbt.2019.49. URL: https://doi.org/10.5441/002/edbt.2019.49.

[126] Adam Seering, Philippe Cudré-Mauroux, Samuel Madden, and Michael Stonebraker. "Efficient Versioning for Scientific Array Databases". In: *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*. 2012, pp. 1013–1024. DOI: 10.1109/ICDE.2012.102. URL: https://doi.org/10.1109/ICDE.2012.102.

[127] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. "Democratizing Data Science through Interactive Curation of ML Pipelines". In: *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. 2019, pp. 1171–1188. DOI: 10.1145/3299869.3319863. URL: https://doi.org/10.1145/3299869.3319863.

[128] Frédéric Simonis. "In-Database Gradient Descent for Machine Learning". Master's Thesis. TUM, Apr. 2017.

[129] Richard T. Snodgrass. "The Temporal Query Language TQuel". In: *ACM Trans. Database Syst.* 12.2 (1987), pp. 247–298. DOI: 10.1145/22952.22956. URL: http://doi.acm.org/10.1145/22952.22956.

[130] Richard T. Snodgrass and Henry Kucera. "Rationale for a Temporal Extension to SQL". In: *The TSQL2 Temporal Query Language*. 1995, pp. 3–18.

[131] Emad Soroush and Magdalena Balazinska. "Time travel in a scientific array database". In: *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. 2013, pp. 98–109. DOI: 10.1109/ICDE.2013.6544817. URL: https://doi.org/10.1109/ICDE.2013.6544817.

[132] Jonas Tappolet and Abraham Bernstein. "Applied Temporal RDF: Efficient Temporal Querying of RDF Data with SPARQL". In: *The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009, Heraklion, Crete, Greece, May 31-June 4, 2009, Proceedings*. 2009, pp. 308–322. DOI: 10.1007/978-3-642-02121-3_25. URL: https://doi.org/10.1007/978-3-642-02121-3_25.

[133] Christina Teflioudi, Faraz Makari, and Rainer Gemulla. "Distributed Matrix Completion". In: *12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012*. Ed. by Mohammed Javeed Zaki, Arno Siebes, Jeffrey Xu Yu, Bart Goethals, Geoffrey I. Webb, and Xindong Wu. IEEE Computer Society, 2012, pp. 655–664. DOI: 10.1109/ICDM.2012.120. URL: https://doi.org/10.1109/ICDM.2012.120.

[134] Manuel Then, Linnea Passing, Nina Hubig, Stephan Günnemann, Alfons Kemper, and Thomas Neumann. "Effiziente Integration von Data- und Graph-Mining-Algorithmen in relationale Datenbanksysteme". In: *Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB, Trier, Germany, October 7-9, 2015*. Ed. by Ralph Bergmann, Sebastian Görg, and Gilbert Müller. Vol. 1458. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 45–49. URL: http://ceur-ws.org/Vol-1458/D06_CRC47_Then.pdf.

[135] Walter F. Tichy. "RCS - A System for Version Control". In: *Softw. Pract. Exp.* 15.7 (1985), pp. 637–654. DOI: 10.1002/spe.4380150703. URL: https://doi.org/10.1002/spe.4380150703.

[136] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. "Rafiki: Machine Learning as an Analytics Service System". In: *Proc. VLDB Endow.* 12.2 (2018), pp. 128–140. DOI: 10.14778/3282495.3282499. URL: http://www.vldb.org/pvldb/vol12/p128-wang.pdf.

[137] Yisu Remy Wang, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. "SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra". In: *Proc. VLDB Endow.* 13.11 (2020), pp. 1919–1932. URL: http://www.vldb.org/pvldb/vol13/p1919-wang.pdf.

[138] Christian Winter, Tobias Schmidt, Thomas Neumann, and Alfons Kemper. "Meet Me Halfway: Split Maintenance of Continuous Views". In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2620–2633. URL: http://www.vldb.org/pvldb/vol13/p2620-winter.pdf.

[139] Carl Witt, Dennis Wagner, and Ulf Leser. "POS: Online Learning for Memory-Aware Scheduling of Scientific Workflows". In: *14th IEEE International Conference on e-Science, e-Science 2018, Amsterdam, The Netherlands, October 29 - November 1, 2018*. IEEE Computer Society, 2018, pp. 399–400. DOI: 10.1109/eScience.2018.00120. URL: https://doi.org/10.1109/eScience.2018.00120.

[140] Jingen Xiang, Huangdong Meng, and Ashraf Aboulnaga. "Scalable matrix inversion using MapReduce". In: *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*. Ed. by Beth Plale, Matei Ripeanu, Franck Cappello, and Dongyan Xu. ACM, 2014, pp. 177–190. DOI: 10.1145/2600212.2600220. URL: https://doi.org/10.1145/2600212.2600220.

[141] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. 2017. arXiv: 1708.07747. URL: http://arxiv.org/abs/1708.07747.

[142] Xiaolong Xie, Wei Tan, Liana L. Fong, and Yun Liang. "CuMF_SGD: Parallelized Stochastic Gradient Descent for Matrix Factorization on GPUs". In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2017, Washington, DC, USA, June 26-30, 2017*. 2017, pp. 79–92. DOI: 10.1145/3078597.3078602. URL: http://doi.acm.org/10.1145/3078597.3078602.

[143] Liqi Xu, Silu Huang, SiLi Hui, Aaron J. Elmore, and Aditya G. Parameswaran. "OrpheusDB: A Lightweight Approach to Relational Dataset Versioning". In: *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 2017, pp. 1655–1658. DOI: 10.1145/3035918.3058744. URL: http://doi.acm.org/10.1145/3035918.3058744.

[144] Ying Yang, Niccolò Meneghetti, Ronny Fehling, Zhen Hua Liu, and Oliver Kennedy. "Lenses: An On-Demand Approach to ETL". In: *Proc. VLDB Endow.* 8.12 (2015), pp. 1578–1589. DOI: 10.14778/2824032.2824055. URL: http://www.vldb.org/pvldb/vol8/p1578-yang.pdf.

[145] Lele Yu, Yingxia Shao, and Bin Cui. "Exploiting Matrix Dependency for Efficient Distributed Matrix Computation". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Ed. by Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives. ACM, 2015, pp. 93–105. DOI: 10.1145/2723372.2723712. URL: https://doi.org/10.1145/2723372.2723712.

[146] Yongyang Yu, MingJie Tang, Walid G. Aref, Qutaibah M. Malluhi, Mostafa M. Abbas, and Mourad Ouzzani. "In-Memory Distributed Matrix Computation Processing and Optimization". In: *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*. IEEE Computer Society, 2017, pp. 1047–1058. DOI: 10.1109/ICDE.2017.150. URL: https://doi.org/10.1109/ICDE.2017.150.

[147] Ying Zhang, Martin L. Kersten, and Stefan Manegold. "SciQL: array data processing inside an RDBMS". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. Ed. by Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias. ACM, 2013, pp. 1049–1052. DOI: 10.1145/2463676.2463684. URL: https://doi.org/10.1145/2463676.2463684.

[148] Martin Zinkevich, Markus Weimer, Alexander J. Smola, and Lihong Li. "Parallelized Stochastic Gradient Descent". In: *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada*. 2010, pp. 2595–2603. URL: http://papers.nips.cc/paper/4006-parallelized-stochastic-gradient-descent.