



TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM School of Computation, Information and Technology

Efficient Analytical Workflows for Computational Database Systems

André Kohn



TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM School of Computation, Information and Technology

Efficient Analytical Workflows for Computational Database Systems

André Kohn

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology 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. Dr. Thomas Neumann
2. Prof. Dr. Alfons Kemper

Die Dissertation wurde am 20.09.2022 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 17.01.2023 angenommen.

Abstract

Demand for data insights is at an all-time high, yet relational data processing integrates poorly into analytical workflows. This thesis improves the workflow capabilities of relational database management systems. First, we introduce the language DashQL, an extension to SQL that describes analytical workflows in a single coherent language. The language facilitates holistic optimizations across the domains of information visualization and traditional relational algebra. We include knowledge about the workflow context to implement visualization-driven aggregation, predicate pushdown, and adaptive materialization. Second, we extend the analytical capabilities of database systems through advanced aggregation functions and statistics. We decouple monolithic aggregation logic into low-level plan operators to compose complex SQL aggregates and speed-up multi-expression queries. Third, we improve the flexibility of compilation-based database systems for varying workflow characteristics. Our adaptive execution framework includes runtime feedback to reduce latencies of cheap queries while reaching maximum efficiency when necessary. Finally, we make analytical workflows more interactive by pushing some computation closer to the user through DuckDB-Wasm. DuckDB-Wasm is a WebAssembly version of the embedded database system DuckDB that evaluates analytical SQL queries efficiently in the browser.

Zusammenfassung

Daten-Analysen sind gefragter denn je, und dennoch ist die relationale Daten-Verarbeitung schlecht in analytische Arbeitsabläufe integriert. Diese Arbeit verbessert die Unterstützung von Arbeitsabläufen in relationalen Datenbanksystemen. Zuerst stellen wir die Sprache DashQL vor, welche die Sprache SQL erweitert, um analytische Arbeitsabläufe zu beschreiben. Die Sprache vereinfacht ganzheitliche Optimierungen in den Domänen der Informations-Visualisierung und der traditionellen relationalen Algebra. DashQL berücksichtigt Kontext-Informationen analytischer Abläufe, um visualisierungs-spezifische Aggregationen, verlagerte Prädikats-Auswertung und adaptive Materialisierung zu implementieren. Außerdem erweitern wir die analytische Funktionalität von Datenbanksystemen um komplexe Aggregatsfunktionen und Statistiken. Wir entkoppeln monolithische Aggregations-Logik in feingranulare Plan-Operatoren, um fortgeschrittene SQL-Aggregate auszuwerten und Abfragen mit mehreren Ausdrücken zu beschleunigen. Weiterhin verbessern wir die Flexibilität kompilierender Datenbanksysteme gegenüber Abläufen mit unterschiedlichen Charakteristiken. Unser adaptives Ausführungs-System benutzt Laufzeit-Informationen, um Latenzen billiger Abfragen zu reduzieren, ohne dabei auf eine maximale Effizienz bei teuren Abfragen zu verzichten. Zuletzt verbessern wir die Interaktivität von analytischen Arbeitsabläufen, indem wir Berechnungen mit Hilfe von DuckDB-Wasm näher zum Benutzer verlagern. DuckDB-Wasm ist eine nach WebAssembly übersetzte Version des integrierten Datenbanksystems DuckDB und kann analytische SQL Abfragen effizient im Browser auswerten.

ACKNOWLEDGMENTS

First, I am very grateful to my advisor Thomas Neumann. Thomas operates at the frontier of database system research, and I feel privileged that I could be a part of his research group. He has developed two of the fastest compilation-based database systems in the world, offering his Ph.D. students an ideal environment for exploring their own ideas. His students are free to follow their interests, which gifted me five years of diverse and self-determined research. I am grateful to Alfons Kemper, who held the first foundational database course in my undergraduate studies. He pulls many strings behind the scenes in our group and gives the students a steady stream of advice and anecdotes. I also want to thank Jana Giceva, who inspired new ideas and assisted us in breaking new ground with research around MLIR.

I am lucky to have worked with Viktor Leis, who introduced me to the research field and convinced me to pursue a Ph.D. He taught me valuable lessons on short evaluation loops, the power of ggplot, and scientific paper writing. Working with him was characterized by explorative discussions with honest feedback, and I warmly recommend him as a mentor.

I want to thank Dominik Moritz for our long chats on data visualization and database systems. Dominik is an open-minded and incredibly well-connected researcher with an impressive background in fast visualization techniques. He helped me shape DuckDB-Wasm and pushed me towards using Arrow for efficient communication with WebAssembly. He also inspired me to try Vega-Lite for declarative visualizations in the language DashQL.

I also want to thank Hannes Mühleisen and Mark Raasveldt for their efforts around DuckDB. They helped me release DuckDB-Wasm as an open-source project and promoted future research around front-facing analytics. Hannes and Mark both have extensive knowledge of vectorized databases, which presented an exciting contrast to compilation-based systems.

I am grateful for my two internships in the Hyper database group at Tableau. My mentor Jan Finis supported me with many technical discussions and gave me the freedom to realize my own ideas during two fascinating projects. I enjoyed working with Michael Haubenschild, Adrian Vogelsgesang, Jonas Kam-



merer, Manuel Then, Jonas Eckhardt, and Tobias Mühleisen, who impressed me through their expertise and work culture.

Thank you also to my colleagues at the TUM, Jan Böttcher, Timo Kersten, Maximilian Bandle, Philipp Fent, Dominik Durner, Mortiz Sichert, Alexander Beischl, Andreas Kipf, Christoph Anneser, Alice Rey, Michael Jungmair, Tobias Schmidt, Altan Birler, Ferdinand Gruber, Bernhard Radke, Lukas Vogel, Michael Freitag, Maximilian Reif, Maximilian Rieger and Alex Khatskevich for debating trends, for evaluating ground-breaking ideas, for enduring euphoric demos, and for all other activities besides our research.

Finally, I want to thank Kristina and my parents for their support and love during this journey. This thesis wouldn't have been possible without you.

Thank You.

Funding.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286).  

PREFACE

Excerpts of this thesis were published in advance.

Chapter 4 has previously been published in:

André Kohn, Viktor Leis, and Thomas Neumann. “Building Advanced SQL Analytics From Low-Level Plan Operators”. In: *SIGMOD*. 2021, pp. 1001–1013

Chapter 5 has previously been published in:

André Kohn, Viktor Leis, and Thomas Neumann. “Adaptive Execution of Compiled Queries”. In: *ICDE*. Apr. 2018, pp. 197–208

André Kohn, Viktor Leis, and Thomas Neumann. “Making Compiling Query Engines Practical”. In: *TKDE* 33.2 (2021), pp. 597–612

Initial experiments were performed in:

André Kohn. “Adaptive Execution of Compiled Queries”. MA thesis. Technical University of Munich, 2017

Chapter 6 has previously been published in:

André Kohn, Dominik Moritz, Mark Raasveldt, Hannes Mühleisen, and Thomas Neumann. “DuckDB-Wasm: Fast Analytical Processing for the Web”. In: *PVLDB*. vol. 15. 12. 2022, pp. 3574–3577

CONTENTS

Acknowledgments	i
Preface	v
1 Introduction	1
1.1 Thesis Statement	2
1.2 Thesis Contributions	2
1.3 Prior Publications and Authorship	4
2 Background	5
2.1 Analytical Data Processing	5
2.1.1 Efficiency through Dynamic Code Generation	5
2.1.2 Towards Computational Database Systems	6
2.2 Information Visualization	7
2.2.1 Declarative Visualization	8
2.2.2 Data Visualization Management Systems	9
3 Complete Analytical Workflows with DashQL	11
3.1 Introduction	11
3.2 Grammar of Analytics	13
3.2.1 SQL Extension	13
3.2.2 Driving Analytical Workflows	16
3.3 Implementation	19
3.3.1 AST Format	19
3.3.2 From AST to Task	20
3.3.3 Adaptive Task Graphs	21
3.3.4 Complementing Vega-Lite	23
3.3.5 Language Extensions	25
3.3.6 Holistic Optimization	26
3.4 Example Data Exploration	29
3.5 Visualization with AM4	32
3.6 Related Work	33
3.6.1 Declarative Analysis Languages	33
3.6.2 Scalable Visual Analysis	35
3.7 Summary	36
4 Evaluating Advanced Analytical SQL Queries	37

4.1	Introduction	37
4.2	Background	39
4.3	From SQL To LOLEPOPs	42
4.3.1	LOLEPOPs	42
4.3.2	From Tree to DAG	44
4.3.3	Advanced Expressions	48
4.3.4	Extensibility	51
4.4	LOLEPOP Implementation	52
4.4.1	Code Generation	52
4.4.2	Tuple Buffer	54
4.4.3	Sorting	56
4.4.4	Aggregation	57
4.4.5	Partitioning	59
4.4.6	Combine	59
4.5	Evaluation	60
4.5.1	Comparison with other Systems	60
4.5.2	Advanced Aggregates in TPC-H	64
4.5.3	LOLEPOPs in Action	65
4.5.4	Adaptive Sorting	66
4.6	Related Work	68
4.7	Summary	69
5	Reducing Latency in Compiling Query Engines	71
5.1	Introduction	71
5.2	Query Execution Via Compilation	73
5.2.1	Latency vs. Throughput Tradeoff	74
5.2.2	Compiling Large Queries	75
5.3	Adaptive Execution	76
5.3.1	Overview	76
5.3.2	Tracking Query Progress	78
5.3.3	Switching Between Execution Modes	78
5.3.4	Choosing Execution Modes	80
5.4	Fast Bytecode Interpretation	82
5.4.1	Virtual Machine	83
5.4.2	Translating into VM Code	84
5.4.3	Register Allocation	86
5.4.4	Linear-Time Liveness Computation	87
5.4.5	Interoperability	91
5.4.6	Optimizations	91

5.5	Evaluation	92
5.5.1	Static vs. Adaptive Mode Selection	93
5.5.2	Adaptive Execution in Action	94
5.5.3	Planning and Compilation Time	96
5.5.4	Performance of Interpreted and Compiled Code	97
5.5.5	Compiling Very Large Queries	99
5.5.6	Adaptivity to Data Size and Parallelism	100
5.5.7	Resistance to Estimation Errors	102
5.6	Related Work	103
5.7	Summary	105
6	Eliminating Latency with WebAssembly	107
6.1	Introduction	107
6.2	Design and Implementation	108
6.2.1	Embedding WebAssembly	108
6.2.2	Web Filesystem	109
6.2.3	Web Workers	110
6.2.4	User-Defined Functions	111
6.3	TPC-H Benchmark	111
6.4	Demonstration Scenario	113
6.5	Summary	114
7	Conclusion	117
7.1	Review of Thesis Contributions	117
7.2	Limitations of the Systems	118
7.3	Future Directions	118
7.3.1	Distributing Workflows	119
7.3.2	Advanced Aggregation	119
7.3.3	Latency-driven Optimization	120
7.4	Concluding Remarks	120
	Bibliography	121

LIST OF FIGURES

Figure 1	Kandel et al. describe five tasks that occur in data analysis workflows [55]. <i>Profiling</i> and <i>Modeling</i> are evaluated within database systems. <i>Discovery</i> , <i>Wrangling</i> and <i>Reporting</i> cannot be fomulated in SQL alone and are often implemented externally.	1
Figure 2	Example of a DashQL script describing a complete analysis workflow. The script loads site activity data, resolves country names via ISO 3166 country codes, aggregates daily page views, and visualizes the data using an area chart and a table. DashQL statements unify user interactions, data loading, and the analysis in a single language.	11
Figure 3	Grammar rules of the five DashQL statements SET, DECLARE, IMPORT, LOAD and VISUALIZE that are combined with the statement rules of PostgreSQL.	14
Figure 4	DashQL scripts as a driver for data analysis workflows. AST nodes store the location in the input text, the node type, the attribute key, the index of the parent node, and either a raw value or a span of children nodes. Script-based analysis workflows allow for interactive exploration, scalable dashboards, and a collaborative workflow development.	16
Figure 5	Example of a task graph that is derived from a previous task graph and an AST-based script difference. The two scripts visualize grouped timeseries data and differ in a deleted statement and the grouping granularity. The AST colors equal statements in green, changes in blue and deletions in orange.	20
Figure 6	Two VISUALIZE statements that produce the same time series line chart, showing website hits of multiple websites. DashQL generates Vega-Lite specifications based on the table schema and statistics.	23

Figure 7	Two load statements that extract two relations from a single JSON document using JMESPath expressions. Both expressions extract populations in Oklahoma. The first expression emits the city data in column-major format, the second expression returns county data in row-major format.	25
Figure 8	M ₄ , a query for value-preserving time series aggregation, described by Jugel et al [54]. This version uses a CTE instead of a subquery with equal semantics.	27
Figure 9	AM ₄ , a more efficient version of M ₄ that provides value-preserving time series aggregation using a single scan and the aggregation functions <code>arg_min</code> and <code>arg_max</code>	28
Figure 10	Authoring an example analysis workflow with DashQL. The workflow explores website activity data in four steps. The steps are labeled with ① to ④ and associate textual changes in the script with adjusted visual output. Visualization statements are colored in green, the input statement and the corresponding predicate in orange.	30
Figure 11	Downloading and rendering dominate the visualization times for increasing data sizes in a client-server setting. M ₄ and AM ₄ efficiently reduce large datasets to a small cardinality that can be visualized quickly.	32
Figure 12	Translation of a GROUP BY operator into a computation graph to construct a DAG of LOLEPOPs.	44
Figure 13	Algorithm to derive the LOLEPOP DAG.	46
Figure 14	Plans for three example queries outlining challenges with composed aggregates, implicit joins and order sensitivity.	48
Figure 15	Plans for three example queries that allow for the optimization of result ordering and aggregate nesting.	50
Figure 16	Plans and simplified code for a query that computes a median, an average, and a distinct sum of two joined relations.	53
Figure 17	Tuple buffer and translator code that accesses sorted key ranges through iterator abstraction at query compile time.	55
Figure 18	Morsel-Driven Quicksort and merge of two partitions. One partition is split eagerly to increase parallelism.	56

Figure 19	Two-phase hash aggregation with two threads. The hash tables on the left are fixed in size while the hash tables in green grow dynamically.	58
Figure 20	Execution times of five TPC-H queries at scale factor 10 with and without additional aggregates.	63
Figure 21	Execution traces of two queries on the TPC-H schema at scale factor 0.5 with four threads and 16 buffer partitions.	66
Figure 22	Sort performance for varying tuple sizes using different access methods.	67
Figure 23	Architecture of compilation-based query engines.	73
Figure 24	Single-threaded query compilation and execution time for different execution modes on TPC-H query 1 on scale factor 1.	74
Figure 25	Execution modes and their compilation times.	76
Figure 26	Illustration of query plan translation to pseudo code. <i>queryStart</i> is the main function. Each of the three query pipelines is translated into a <i>worker</i> function. The lower left corner shows that the work of each pipeline is split into small morsels that are dynamically scheduled onto threads.	77
Figure 27	Switching on-the-fly from interpretation to execution. The dispatch code is run for every morsel.	79
Figure 28	LLVM compilation time for (un-)optimized machine code for TPC-H and TPC-DS queries.	80
Figure 29	Extrapolation of the pipeline durations.	81
Figure 30	VM code fragment implementing the interpreter loop. <i>ip</i> points to the current instruction and <i>reg</i> points to the memory storing the registers.	82
Figure 31	Translation of LLVM IR into VM code.	84
Figure 32	Computing the liveness of a variable <i>x</i> . The vertices are basic blocks, which are connected by control flow edges (i.e., branch instructions).	87
Figure 33	Linear-time algorithm for liveness computation.	88
Figure 34	Dominator tree annotated with pre-/post-order.	89
Figure 35	Geometric mean of all TPC-H queries including planning, compilation, and execution using 8 threads for different scale factors and execution modes.	93

Figure 36	Plan and execution trace of TPC-H query 11 on scale factor 1 using 4 threads. The optimized mode is not shown, as its compilation takes very long (103ms).	95
Figure 37	Compilation times of queries with a large number of instruction using optimized compilation, unoptimized compilation and interpretation.	99
Figure 38	Plan and execution traces of TPC-H query 4 for adaptive execution on scale factors 0.25 and 1 using 1, 2, and 4 threads.	101
Figure 39	The effect of different speed-up factors between the bytecode interpreter and unoptimized compiled code on the execution time for the TPC-H queries 1 – 5 at the scale factors 1 and 10 using 8 threads. The constant used in our system is marked in blue.	102
Figure 40	Browser-based analytics tools process data either locally with a low efficiency or on servers with a high latency. DuckDB-Wasm pushes the boundaries with fast analytical processing for the Web.	107
Figure 41	A SQL script that downloads stock data from AWS S3 stored in a Parquet file and joins it with a portfolio stored in a CSV file. The left side presents multiple ways to execute the script in a distributed setting. ① shows the traditional separation between client and server, ③ a fully local execution, ② a hybrid mode in between.	109
Figure 42	A shell that runs entirely in the browser and evaluates SQL queries using DuckDB-Wasm. The figure shows a query joining two parquet files with the relations <i>orders</i> and <i>customer</i> of the TPC-H benchmark at scale factor 0.1. ① lists the query results and page accesses, ② shows the query plan.	113

LIST OF TABLES

Table 1	LOLEPOPs for advanced SQL analytics. The input and output are either a tuple stream (▶, ▯) or tuple buffer (▮, ▯).	40
Table 2	Execution times in seconds of queries with simple aggregates in HyPer, PostgreSQL and MonetDB.	61
Table 3	Execution times in seconds for advanced SQL queries on the TPC-H lineitem table (scale factor 10).	62
Table 4	Planning and compilation times in ms for TPC-H queries on PostgreSQL (“PG”), MonetDB (“Monet”), and HyPer.	96
Table 5	Execution times of TPC-H queries on scale factor 1 on PostgreSQL (“PG”), MonetDB (“Monet”) and HyPer. The geometric means (“geo.m.”) are over all 22 queries.	97
Table 6	CPU counters ($\times 10^6$) for TPC-H queries 1 and 5 on scale factor 1 using 1 thread.	98
Table 7	Execution times in seconds for TPC-H queries at the scale factors 0.01, 0.1 and 0.5.	112

1 | INTRODUCTION

The amount of data that is being collected has never been greater. This has a transformative impact on organizations, as they slowly understand the value of data-driven decision-making across all departments and roles. Part of the transformation is the data analyst that faces new requirements while growing into an integral component of modern business operations.

In [55], Kandel et al. conducted a study on current data analysis practices in enterprises. They characterize industrial methodologies based on interviews with analysts from 25 organizations across multiple sectors, including health-care, retail, marketing, and finance. The study identifies the five high-level tasks shown in Figure 1 that frequently occur in analysis workflows: *Discovery* refers to locating the data, such as files stored in a cloud data store. *Wrangling* means extracting tuples from that data, for example, by interpreting it as JSON or CSV. *Profiling* verifies the quality of the data by checking distributions and searching for errors and outliers. *Modeling* transforms the data, for example, by computing summary statistics. *Reporting* produces insights that can be communicated to consumers of the analysis.

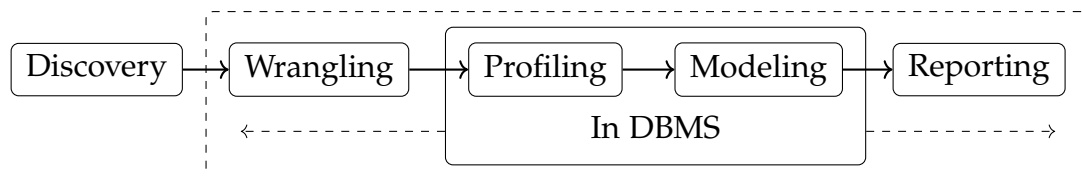


Figure 1: Kandel et al. describe five tasks that occur in data analysis workflows [55]. *Profiling* and *Modeling* are evaluated within database systems. *Discovery*, *Wrangling* and *Reporting* cannot be formulated in SQL alone and are often implemented externally.

The study further reveals common pain points. First, operating these workflows in a repeatable, reliable, and scalable way is difficult as the task execution is spread across several systems. These systems produce different intermediate results, such as scripts, spreadsheet formulas, and data sets, that cannot be assembled quickly. Additionally, the first two tasks, *Discovery* and *Wrangling* are the most tedious and time-consuming. They involve locating and ingesting

incomplete data from multiple data sources and in different formats, a task underserved by current systems. Moreover, the interviewed analysts consider today's visualization tools to be insufficiently scalable and too static for *Reporting* as they lack access to data provenance.

We consider the fractured task evaluation a significant obstacle to efficient analysis workflows. Ultimately, analysis workflows are processes for ingesting, transforming, and visualizing data sets, and yet, database management systems drive only two out of the five tasks. *Profiling* and *Modeling* can be formulated in SQL and therefore draw on decades of research on efficient query optimization and execution. *Wrangling* and *Reporting* both interact with database systems but are not expressible with SQL alone and are thus often implemented externally. However, external implementations usually give up on holistic data-driven optimizations, leaving the question if we should not expand the workflow capabilities of database systems instead.

1.1 THESIS STATEMENT

This thesis hypothesizes that relational database management systems can efficiently evaluate complete analysis workflows. We suggest using a coherent SQL-based language model for workflows to enable optimizations across the domains of information visualization and data processing with traditional relational algebra. Such a language allows for a combination of advanced database functionality, such as complex aggregation functions and declarative visualization specifications. Finally, we increase the sensitivity of database systems towards query latencies in interactive workflows through adaptive execution strategies and by pushing computation to the user.

1.2 THESIS CONTRIBUTIONS

This thesis makes contributions to accelerate analysis workflows in relational database management systems. We call the resulting systems computational databases and zoom in on four aspects that target the language model, aggregation, latency, and distribution of workflows.

COMPLETE ANALYTICAL WORKFLOWS WITH DASHQL. Chapter 3 introduces the language DashQL. DashQL combines SQL, the grammar of relational database systems, with a grammar of graphics in a grammar of analytics. It supports preparing and visualizing arbitrarily complex SQL statements in a single coherent language. The proximity to SQL facilitates holistic optimizations of analysis workflows covering data input, encoding, transformations, and visualizations. These optimizations use relation and query metadata for visualization-driven aggregation, predicate pushdown, and adaptive materialization. We introduce the DashQL language as an extension to SQL and describe the efficient and interactive processing of text-based analysis workflows.

ADVANCED ANALYTICS USING LOW-LEVEL PLAN OPERATORS. Aggregation and statistics are prime sources of complexity in analysis workflows, and Chapter 4 describes how to evaluate them efficiently. SQL already offers various functionalities to summarize data, such as associative aggregates, distinct aggregates, ordered-set aggregates, grouping sets, and window functions. These features add significant complexity to a relational database system, restraining the support for additional sophisticated and combined aggregation functions. Chapter 4 proposes a unified framework for advanced statistics that composes all flavors of complex SQL aggregates from low-level plan operators. These operators can reuse materialized intermediate results, which decouples monolithic aggregation logic and speeds up complex multi-expression queries. Our framework modularizes aggregate implementations and outperforms traditional systems whenever multiple aggregates are combined.

REDUCING LATENCY IN COMPILING QUERY ENGINES. Analytical workflows are sensitive to latency spikes in interactive settings, which poses a problem for database systems that use dynamic code generation for executing queries. These systems compile SQL to machine code, resulting in very efficient query plans at the cost of increased base latencies. Generating machine code can take hundreds of milliseconds for complex queries, even with fast compilation frameworks like LLVM. Such durations can disadvantage workflows that execute many complex but quick queries. To solve this problem, Chapter 5 introduces an adaptive execution framework, which dynamically switches from interpretation to compilation. The framework uses a fast bytecode interpreter for LLVM that executes queries without costly translation to machine code and dramatically reduces query latency. Adaptive execution is fine-grained

and executes code paths of the same query using different execution modes. We show that this approach achieves optimal performance in a wide variety of settings—low latency for small data sets and maximum throughput for large data sizes.

ELIMINATING LATENCY WITH WEBASSEMBLY. Chapter 6 introduces DuckDB-Wasm. DuckDB-Wasm is a WebAssembly version of the embedded database system DuckDB that evaluates analytical SQL queries in the browser. It can further reduce query latencies of analysis workflows by eliminating costly round-trips over the internet and pushing computation closer to the user. The database runs asynchronously in web workers, supports efficient user-defined functions written in JavaScript, and features a browser-agnostic filesystem that reads local and remote data in pages. We show that DuckDB-Wasm outperforms previous data processing libraries for the Web in the TPC-H benchmark at multiple scale factors.

1.3 PRIOR PUBLICATIONS AND AUTHORSHIP

Although I am the principle author of the research in this dissertation, all of the work was done in collaboration with my advisor Thomas Neumann. Chapter 3 introduces the language and system DashQL that I discussed extensively with Dominik Moritz from the Carnegie Mellon University. Chapter 4 proposes low-level plan operators for advanced aggregates and was published at ACM SIGMOD 2020 as joint work with Viktor Leis. Chapter 5 describes the adaptive optimization of query latencies in compiling query engines that I started to explore in my Master's thesis in 2017 and later expanded on at IEEE ICDE 2018 and IEEE TKDE 2019 together with Viktor Leis. Chapter 6 introduces DuckDB-Wasm that I developed with Dominik Moritz as well as Mark Raasveldt and Hannes Mühleisen from the Centrum Wiskunde & Informatica and was published at VLDB 2022. In this thesis, I use the first person plural to reflect my collaborators' contributions.

2 | BACKGROUND

This thesis contributes to the intersection between data processing and information visualization and builds on prior work in both domains. We outline foundational research for the thesis in the following and introduce specific related work in each chapter.

2.1 ANALYTICAL DATA PROCESSING

In 1970, Codd introduced the relational data model [25]. He set off a race for building the fastest database management systems, processing what would become the de-facto standard representation of structured data. The race is still ongoing today and is fueled by the continuously evolving hardware and software landscapes. Particularly online analytical data processing, also called "OLAP" presents an attractive target for low-level optimizations that squeeze the last bit of performance out of given hardware. OLAP queries involve joining and summarizing large datasets, an operation that quickly dominates the entire execution time of analytical workflows.

2.1.1 Efficiency through Dynamic Code Generation

Around the beginning of the 20th century, the rise of main memory capacities powered a new wave of database systems entering this race. Large main memory sizes steered database research from shadowing costly disk accesses to optimizing for swift CPU caches. Boncz et al. demonstrated the severe impact of cache-aware algorithms in the database systems MonetDB and VectorWise [81, 15]. In 2011, Neumann pioneered a novel execution strategy for the database system HyPer that efficiently translates SQL queries to machine code [87]. This dynamic code generation eliminates the traditional overhead of interpreting query plans by flattening plan pipelines into compact loops. The generated code keeps values in CPU registers as long as possible and even rivals the efficiency of hand-written query programs. Ever since, code gener-

ation has established itself as a major driver for fast analytical processing in database systems and is used by a large and growing number of commercial systems (e.g., Hekaton [31, 37], MemSQL [90], Spark [2], and Impala [124]) as well as research projects (e.g., HIQUE [68], DBToaster [60], Tupleware [28, 27], LegoBase [59], ViDa [56], Vodoo [95], Weld [89], Peloton [82, 92]).

On the flip side, the increased efficiency is accompanied by a higher system complexity as code generation only serves as a staging point into the territory of classical compiler construction. While chasing maximum performance, database systems slowly morph into domain-specific compilers with custom intermediate representations and own optimization passes. The framework LLVM provides initial relief through modular access to compiler components but only partially fulfills the requirements of a database system. Kersten et al., therefore introduce their code generation framework for the database system Umbra that abstracts operator translators, data structures, tuple handling, and SQL values [57]. The system LegoBase reduces the system complexity of the *low-level* code generation by lowering high-level programs through multiple but simple, intermediate representations [59, 111]. LB2 simplifies code generation by deriving query programs automatically from interpreting operators written in the high-level language Scala [118].

2.1.2 Towards Computational Database Systems

The growing SQL standard is an additional driver for continuous research on relational database systems. Since the '92 standard, SQL has grown considerably and offers a variety of features for analytical workflows. The statistics functionality in the standard started with simple associative aggregates such as SUM, COUNT, MIN and MAX that can be computed for grouped data. In SQL:1999, the grouping of values was extended by grouping sets that allow summarizing data sets at multiple aggregation granularities simultaneously. SQL:2003 added *window functions* that compute aggregates for individual rows based on surrounding values in a partition. They simplify time series analysis and provide a convenient way to express ranking, moving averages, and cumulative sums. SQL:2003 also added the capability of computing percentiles (e.g., median). PostgreSQL calls these functions *ordered-set aggregates* because percentiles require (partially) sorted data.

In parallel to the standardization efforts, several projects have extended relational database systems to integrate external logic of analysis workflows. In-

egratedML by De Boe et al. [30] extends the SQL language with the statements `CREATE MODEL`, `TRAIN MODEL` and the functions `PREDICT` and `PROBABILITY` that add machine learning capabilities to SQL scripts. SolveDB, by Siksnyš et al. [114] introduces the statement `SOLVESELECT` for efficient problem solving with linear programming in SQL. Hellerstein et al. implemented MADlib [49], a rich library for the databases PostgreSQL and Greenplum that provides advanced analytical functions for machine learning and statistics. Yu proposes the statement `CREATE GEOVIZ`, implementing spatial optimizations for data preparation and map visualizations. Schüle et al. describe ArrayQL [107], a SQL extension with improved support for multidimensional arrays. Edi-Flow [12], by Benzaken et al., is a workflow platform for visual analytical applications that implements a process model on top of relational algebra. Howe et al. describe the merits of the SQL language over R programs for collaborative science workflows in their system SQLShare [52]. The project Pipemizer [40], by Gakhar et al., lifts relational optimizations to the level of analytical workflows by reordering jobs and identifying common subexpressions. We identify this as a trend towards computational database systems that combine the efficient processing of traditional relational algebra with domain-specific algorithms.

2.2 INFORMATION VISUALIZATION

Three years prior to Codd's publication of the relational data model, the french cartographer Jacques Bertin made a major contribution to the theory of information design in 1967. His *Semiology of Graphics* describes and categorizes foundational concepts for graphic communication [13]. He focuses on our visual perception of data with concepts like retinal variables that describe the position, size and shape of a graphical representation. 16 years later, Tufte expanded the field with an extensive collection of statistical graphics and a detailed analysis on how to display data [123]. He introduced the concept of data-ink ratio and coined the term chart junk as superfluous visual elements that distract viewers from the actual information. Bertin and Tufte both analyzed the effectiveness of graphical representations and provided a foundation for following research on information design.

2.2.1 Declarative Visualization

In 1999, Wilkinson shaped the field of declarative visualization specifications decisively through his work *Grammar of Graphics* [129]. Wilkinson adopts the work of Bertin and proposes a comprehensive language to construct a wide range of statistical graphics from individual components. The result is a formal model that allows users to describe a statistical chart instead of selecting from a set of predefined chart types.

Inspired by Wilkinson, Stolte et al. later published the Polaris system that constructs visual specifications of graphical displays based on pivot tables [116]. Polaris adopted the work of Wilkinson to formalize statistical transformations and graphical representations but deviated from it in the underlying data model. Wilkinson describes data transformations as part of his own non-relational table algebra, while Polaris builds on a relational model for compatibility with SQL. Through SQL, Polaris provides data insights for a large variety of OLAP databases later popularized in their commercial offering, Tableau.

In 2010, Wickham published `ggplot2` [127], a grammar of graphics for the programming language R that extends Wilkinson's ideas with a component hierarchy [128]. Wickham proposes to change the parameterization of Wilkinson's components to build graphics from multiple layers of data. These layers increase the interoperability and reuse between components and enable hierarchical default settings that condense specifications. `ggplot2` is embedded into the language R to reduce the friction between data visualization and the rest of the analysis workflow. According to Wickham, using R offers a wide range of prepackaged statistics functions and facilitates data import and manipulation. This thesis follows a very similar idea in Chapter 3 by embedding visualization and data import into the language SQL.

Polaris and `ggplot2` share the same goal of abstracting and simplifying graphical configuration settings. The projects D3 [16] and Vega [105] contrast this approach by providing lower-level interfaces for visualization design. They offer fine-grained control over statistical charts at the cost of more specific declarations. D3 is a visualization library that targets the document object model (DOM) in web browsers. It exposes arbitrary document elements instead of hiding them behind a scene graph, improving the integration into the programming language JavaScript. Vega, on the other hand, specifies visualizations entirely in JSON documents. These documents are parsed and translated into optimized dataflow graphs that unify input data and scene graph elements. Both have in common that constructing a statistical graphic can become ver-

bose. As a result, several projects build on top of Vega and D3 and use them as intermediate representations [23, 75, 113, 42, 132].

One such project is Vega-Lite, a higher-level grammar on top of Vega that Wongsuphasawat et al. developed for the Voyager system [131, 104]. Vega-Lite abstracts the Vega grammar to simplify the enumeration of designs and optimize data transformations and visual encodings. Their compiler lowers Vega-Lite to Vega and uses a rule-based system to resolve ambiguities in specifications. They also describe the Compass recommendation engine that generates Vega-Lite specifications from user selections, the data schema, and statistical properties. Vega-Lite trades general expressiveness of the grammar for a convenient formalism and more room for optimizations. We adopt this concept in Chapter 3 with the DashQL visualization statement that embeds Vega-Lite specifications into SQL scripts.

2.2.2 Data Visualization Management Systems

Wu et al. outline their vision of an integrated Data Visualization Management System (DVMS) that uses a declarative language to compile end-to-end visualization pipelines into a set of relational algebra queries [133]. They note that current visualization tools decouple the data processing from computations targeting visualization and rendering. These tools spare costly roundtrips to database systems through local result caching, and additional downstream data transforms. This introduces an information gap towards database systems that are no longer aware of the user context. Instead, they waste computing resources when repeatedly re-evaluating complex queries on user interactions. Wu et al. use the panning of a map as an example, where a small gesture may issue a query to recompute the entire map, even if the new input affects only a tiny fraction of the result set. They propose to bridge this gap by executing common visualization transforms directly in the database. Their vision hints at novel visual optimization techniques that may reduce rendering latencies through occlusion filters, output-based downsampling, and distributed rendering. This thesis develops this vision in Chapter 3 by extending the SQL language with new statements for visualization, interaction, and data loading. We call this language DashQL, a declarative language for DVMS's, and implement optimizations such as visualization-oriented time-series aggregation and projection pushdown.

3

COMPLETE ANALYTICAL WORKFLOWS WITH DASHQL

3.1 INTRODUCTION

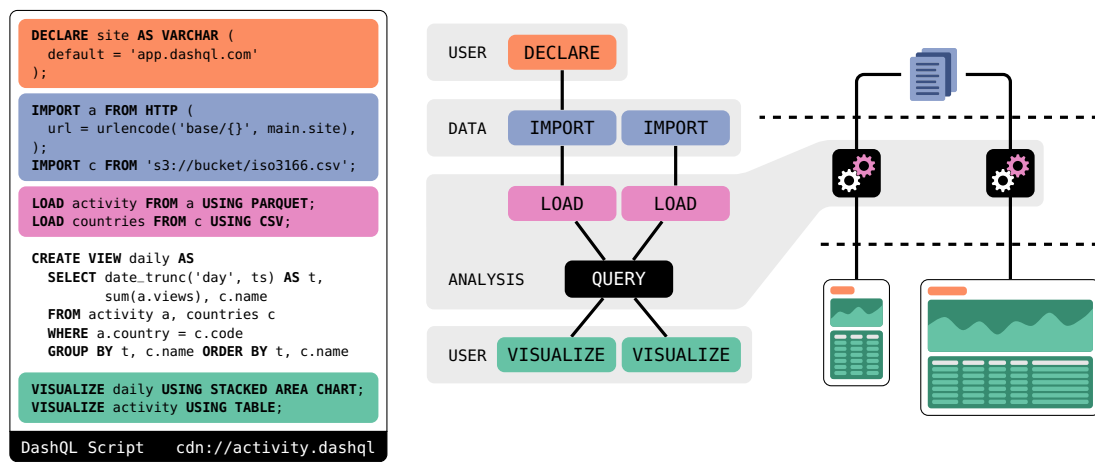


Figure 2: Example of a DashQL script describing a complete analysis workflow. The script loads site activity data, resolves country names via ISO 3166 country codes, aggregates daily page views, and visualizes the data using an area chart and a table. DashQL statements unify user interactions, data loading, and the analysis in a single language.

Interactive Data Analysis has evolved as an umbrella term for diverse research around approachable, inspirational, explanatory, and efficient data processing. Decades of prior work in these areas have assembled a comprehensive toolbox, guiding users on their paths toward valuable data insights. A common principle among these tools has been the unification of graphics and database interactions, creating a gap towards database query languages like SQL. Pioneering systems like Polaris shield users from database specifics by pairing a graphic taxonomy with an own relational table algebra [116]. This table algebra is lowered to SQL transparently, which abstracts from subtle differences between SQL dialects and allows supporting various database systems through a single interface.

In the meantime, however, SQL has become the de-facto standard for data transforms at all scales, ranging from embedded systems (e.g., DuckDB [100],

SQLite [47]) to large data warehouses (e.g., Snowflake [29], F1 [102], Procella [22], Presto [110], Redshift [45], Azure Synapse [3], CockroachDB [117], Hive [18]). Therefore, database abstractions that do not match SQL in expressivity turn into an explicit translation layer that data analysts might have to work around. This is particularly pronounced for advanced SQL functionality such as nested subqueries, non-inner joins, window aggregates, or grouping sets that are often omitted as early victims during generalization. A lack of these features can render today's tools insufficient for analysts that use SQL as their mental model for database interactions.

Additionally, database abstractions prevent holistic optimizations of analysis workflows. The optimization of SQL queries is a well-studied problem but is usually unaware of how data is ingested and how the results are consumed [11]. Data analysts, therefore, propagate information back into the database, for example, by optimizing requests for the following visualization. This is not only error prone but exposes relational optimizations to the user. Database abstractions also obfuscate the capabilities of the underlying database. It is not uncommon in today's analysis tools to prepare and cache volatile data to optimize the repeated and interactive query evaluation during exploration [120]. However, this is a pitfall as it nullifies standard database optimizations like projection and selection pushdown. Structured file formats like Parquet allow reading data partially based on the query columns and filters. A tool that lacks these query-driven optimizations might therefore be *slower* if it loads *unnecessary* data for a workflow.

We expand the vision of Wu et al. [133] and propose a language for a Data Visualization Management System (DVMS) that embeds data retrieval, loading, and visualization into SQL. We call this SQL dialect DashQL and explain how a single coherent language model can drive interactive analysis workflows. Figure 2 shows the first example of a DashQL script that visualizes grouped timeseries data using an area chart and table. In the figure, the input script on the left is translated to a graph of tasks that drives the parallel evaluation of statements. The right side of the figure hints at the visual output of the script as an interactive dashboard including an input field at the top of the screen, followed by the two visualizations.

The contribution of this chapter is twofold. We first introduce the language grammar and statement semantics in Section 3.2 and outline how a SQL dialect facilitates interactive exploration, scalable dashboards, and workflow development. We then describe the efficient evaluation of DashQL workflows in Section 3.3 and present holistic optimizations that use metadata for predicate

pushdown and adaptive materialization. Section 3.3 also introduces AM₄, an optimization in DashQL that accelerates visualizations with time series data. We demonstrate DashQL examples throughout the chapter and author an interactive analysis workflow step-by-step in Section 3.4. We measure the performance of the holistic optimization AM₄ in Section 3.5. We close with a discussion of related work in Section 3.6 and a summary of the chapter in Section 3.7.

3.2 GRAMMAR OF ANALYTICS

DashQL unifies the predominant grammar of relational [26] database systems, SQL, with a grammar of graphics [129] into a grammar of analytics. This section introduces the DashQL language and its role in an analytics system. We first list the grammar rules of DashQL and describe the semantics of every new statement. Afterward, we present three advantages of driving analysis workflows with a coherent analysis language.

3.2.1 SQL Extension

DashQL introduces the five statements **SET**, **DECLARE**, **IMPORT**, **LOAD** and **VISUALIZE** to the SQL language. Together, they extend SQL just enough to specify where data is located, how it can be loaded, and how it can be visualized for users. This allows DashQL to describe complete analysis workflows in self-contained scripts while preserving the expressiveness of arbitrary SQL queries. The grammar rules of all statements are shown in Figure 3 and are outlined in the following.

SET is a utility statement that defines global script properties as individual key-value pairs. This allows modifying script evaluation settings or provide script metadata such as titles, descriptions or versions.

DECLARE declares values that are provided to the script at runtime. For example, an **DECLARE** statement with identifier *x* and value type **FILE** presents an input control to users that opens a file picker dialog when clicked. The provided file is then exposed to the remainder of the script through the identifier *x*. **DECLARE** may further be followed by an explicit component type and configuration options, matching additional settings like default values.

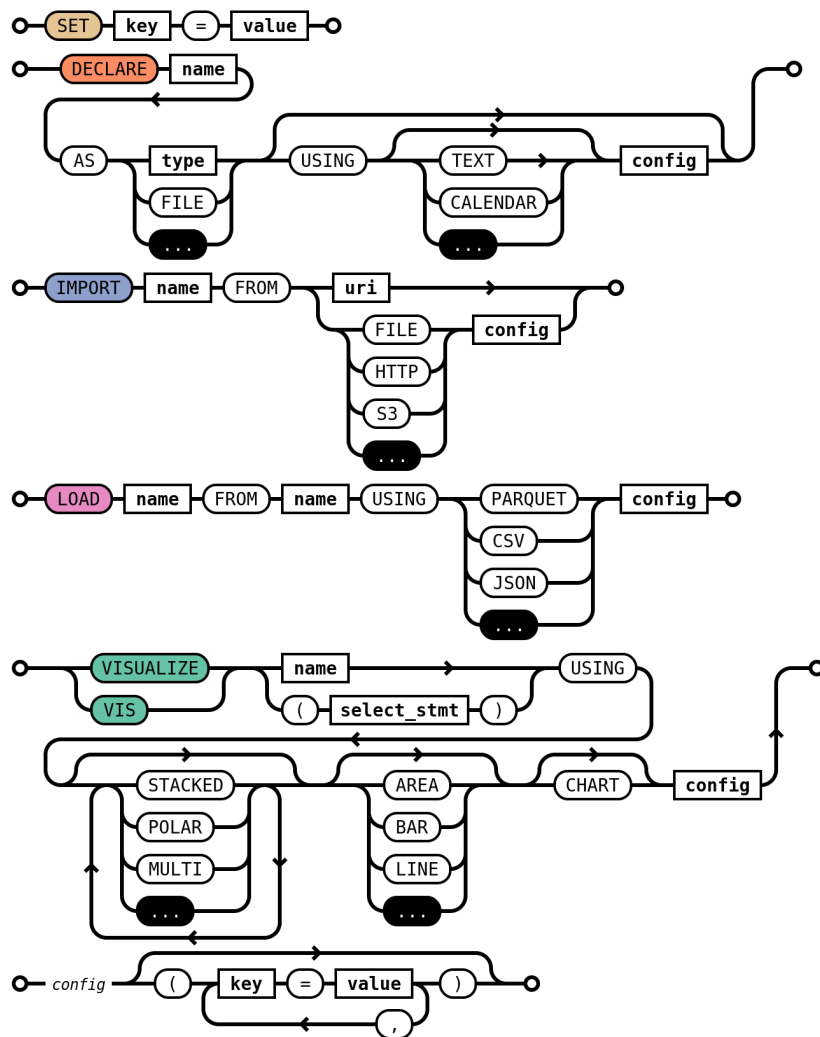


Figure 3: Grammar rules of the five DashQL statements SET, DECLARE, IMPORT, LOAD and VISUALIZE that are combined with the statement rules of PostgreSQL.

IMPORT accompanies DECLARE as the second statement that provides input data for DashQL scripts. In its simplest form, the rule `IMPORT name FROM uri` specifies a raw data source as a single URI. The value `"https://a/b.parquet"`, for example, declares a remote file that will be loaded using HTTPS. If a simple URI is not sufficient, the statement may alternatively be written with the explicit keyword `HTTPS` followed by fine-granular settings such as the method type or request headers. Similarly, the `IMPORT` statement allows supporting additional source types such as AWS S3, following the same syntax.

The fourth statement **LOAD** defines how raw data can be loaded into the database. We deliberately separate the importing of opaque data and the extraction of relations since the boundary between these two can be fuzzy and

depends on the capabilities of the underlying database. DuckDB, for example, can partially scan remote Parquet files over HTTPS using a dedicated table function and a virtual file system abstraction. This collapses both statements into following SQL statements, emphasizing the decoupled nature of declarations in a script and their efficient execution. Other databases without these capabilities might need to execute either one or both tasks explicitly upfront. Similar to `IMPORT`, the `LOAD` statement can also be extended with additional keywords to introduce new data formats to the language.

The last statement `VISUALIZE` displays data using charts and tables. Creating visualizations is an iterative process that benefits from short round-trip times between ideas and their realizations. DashQL offers approachable and fast exploration by combining a simple and short syntax with a fallback to a full grammar of graphics for refinements. After all, visualizations are created in tandem with SQL statements which already provide useful information such as the attribute order of SQL projections or data types.

For example, users might want to display timeseries data with a time attribute `t`, and a value attribute `v` backed by a SQL query such as `CREATE TABLE a AS SELECT t, v ...`. Creating the first visualization for this table can be as simple as `VISUALIZE a USING LINE`. Without further information, DashQL assumes that `t` and `v` were deliberately provided as first and second attributes referring to `x` and `y` values of the line chart. Alternatively, SQL column aliases can be used in anticipation of ambiguities to name `x` and `y` explicitly, as in `SELECT v AS y, t AS x ...`. Later iterations may then add a third attribute in the SQL projection list followed by `VISUALIZE a USING MULTI LINE`, in which case the new attribute would map to the line color. This example shows that the interplay with traditional SQL statements may provide enough information to simplify the syntax significantly through carefully chosen defaults.

This simplified syntax enables rapid prototyping but may not suffice for advanced analysis reports. The grammar therefore supports an alternative grammar rule with `VISUALIZE a USING (...)` that describes visualizations using raw Vega-Lite specifications. In fact, the *simple* form is lowered down to Vega-Lite internally, enabling automatic rewrites as explicit specification when refinements are needed. In alignment with SQL, the Vega-Lite attributes are case insensitive in DashQL and JSON objects are replaced with nested key-value pair lists enclosed by round brackets. The systematic generation and completion of Vega-Lite specifications is described in Section 3.3.4.

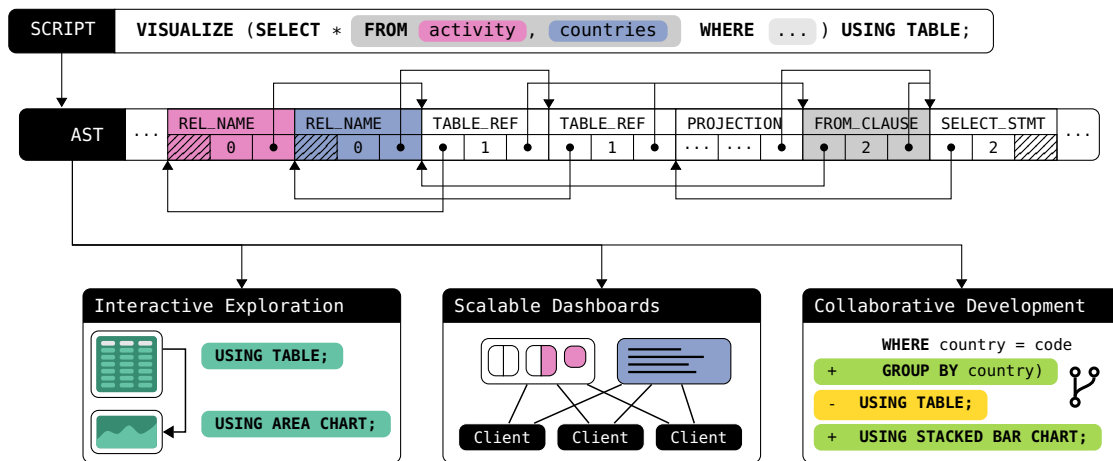


Figure 4: DashQL scripts as a driver for data analysis workflows. AST nodes store the location in the input text, the node type, the attribute key, the index of the parent node, and either a raw value or a span of children nodes. Script-based analysis workflows allow for interactive exploration, scalable dashboards, and a collaborative workflow development.

3.2.2 Driving Analytical Workflows

An extended SQL dialect offers an opportunity to describe entire analysis workflows in self-contained scripts. These scripts become the single source of truth for an analysis system and can drive features such as interactive exploration, scalable dashboards, and collaborative workflow development. Figure 4 illustrates these features based on a single visualization statement that displays the results of a join as a table.

Interactive Exploration

DashQL demystifies system internals by replacing a multitude of configuration knobs with guided textual editing. DashQL strikes a balance between flexibility and intuition by providing short and long versions of the different grammar rules. This flexibility allows users to start the exploration with short statements and later refine the workflow by manually adjusting inferred properties. This simplifies the exploration as the syntactical differences between statements stay small.

The example in Figure 4 expresses the intent to display data as a table by writing `VISUALIZE ... USING TABLE`. The short syntax allows altering the visualization quickly. For example, a user can replace the keyword `TABLE` with `AREA CHART` to change the visualization type and add the keyword `STACKED` to

group areas based on an additional attribute. Once the correct chart type is found, a user can adjust fine-granular configuration options such as colors and labels through explicit Vega-Lite settings. These rewrites can either be done manually by changing the script text or by modifying a previously rendered chart. The result is an interactive loop where partially evaluated DashQL workflows guide users through following refinements.

Additionally, the analysis workflows are interactive themselves through the DECLARE statement. These statements parameterize workflows explicitly by exposing variables to viewers. This allows embedding arbitrary complex SQL queries into the workflow and steer them through input controls. A popular alternative to DECLARE statements is to derive raw SQL text from input values through text interpolation. This is flexible, but complicates the semantic analysis of workflows as it gives up crucial information about statement dependencies, types, and the exact usage of parameters. It also requires a preprocessing step to generate the actual script text which does not align well with the continuous and iterative re-evaluation of workflows.

DashQL distinguishes between the analysts authoring workflows and the viewer that consume the workflows's output. Viewers are not exposed to the language but instead only see the results from statements as opaque analysis dashboards. Authors, in contrast, see the language and visual output side-by-side and benefit from semantic information in the script editor. Interactive exploration in DashQL is therefore emphasized differently for these two user groups as authors benefit from frictionless feedback loops for textual changes while viewers require efficient re-evaluations after changing input values.

Scalable Dashboards

DashQL simplifies the sharing of analysis dashboards. A workflow is a single self-contained script text and can be treated as such for the distribution to multiple users. This decouples the workflow description from the evaluating analysis tool, similarly to SQL being the common denominator between relational database systems. Sharing a data analysis workflow is *cheap* since there is no dependency on specific service resources except for the workflow's input data. The price for serving this data is often *lower* than maintaining computing resources for traditional server-based analytics tools, more so with scalable cloud storage services and large content delivery networks.

We introduce the language DashQL alongside a reference implementation available at github.com/ankoh/dashql. The implementation is powered by

DuckDB-Wasm, an efficient WebAssembly version of the analytical database DuckDB [100] that we describe in Chapter 3. It evaluates entire analytical workflows ad-hoc in the browser, presenting a cost-efficient and interactive solution without dedicated analytics servers. The lack of a dedicated server increases the horizontal scalability of the system at the cost of higher bandwidth requirements for the viewers. According to Vogelsgesang et al, shared analysis workflows on smaller datasets are not uncommon today [125]. They state that only approximately 600 out of 62 thousand workbooks uploaded to the service Tableau Public contain more than a million tuples. All other workflows fall into browser-manageable data sizes, eliminating the need for dedicated computing resources in the cloud.

This also covers workflows that process larger datasets but reduce the data size quickly based on user input. For example, if a workflow processes event data of a logging service, the entire dataset for all users might easily exceed petabytes of records. However, if the workflow analyzes specific user events over a fixed period, the datasets can get sufficiently small. Section 3.3.6 introduces holistic optimizations that optimize the amount of loaded data based on SQL queries in the workflow. Nevertheless, the language DashQL itself is not limited to small datasets. It instead offers an opportunity to dynamically combine client and server-side implementations to optimize for scalability and interactivity wherever possible and fall back to traditional server-side processing when needed.

Collaborative Development

DashQL also simplifies the collaborative development of analysis workflows. Text-based version control systems like *Git* dominate distributed software development today. Since DashQL workflows are self-contained scripts, they can be developed as part of a versioned development process. Users can fork DashQL workflows and contribute changes back through simple textual updates. This process is facilitated by the concise grammar of SQL that keeps the textual differences small. Figure 4 demonstrates this versioning by adding a grouping clause and a changed chart type in the example statement. The patch tracks the new grouping by the country attribute and the visualization as a stacked bar chart in the same script. DashQL workflows can therefore be created, updated, forked, and discussed in environments that have already proven their effectiveness in collaborative development.

3.3 IMPLEMENTATION

This section outlines the implementation of a DashQL-powered analysis tool. We first describe the efficient AST encoding we use as a textual language model for analysis workflows. This model allows the runtime to update only the parts of the execution state that have changed instead of complete re-evaluations. We then introduce the concept of tasks and show how adaptive task graphs can be maintained using immediate difference computations. We discuss the extensibility of DashQL and the use of query metadata to simplify declarative visualizations for fast exploration. Finally, we present two examples of holistic optimizations accelerating the coupled workflow components.

3.3.1 AST Format

DashQL translates many user interactions into modifications of the associated script text. This positions the underlying text model as a fundamental component of the entire system. Our implementation is therefore built around a fast syntactical analysis backed by an efficient representation of the abstract syntax tree (AST). The parser extends the SQL grammar rules of PostgreSQL and allocates compact AST nodes into a single, bump-allocated memory buffer. This accelerates parsing and increases the cache efficiency of any following operations, such as tree traversals. An AST node is exactly 20 B large and stores the location in the input text, the node type, the attribute key, and the index of the parent node. It also stores either a raw integer value or a span of children nodes in the same buffer. The text location associates each node with the substring matched by its grammar rule, enabling partial rewrites of individual statements. The AST further acts as an auxiliary data structure and references string literals in the original script text instead of copying them. Children of an AST node are further stored in sorted order based on the attribute key, accelerating key lookups and recursive comparisons.

Figure 4 illustrates the AST encoding of an example statement that visualizes an inline SQL query joining two base relations. The AST presents two nodes of type `REL_NAME` that match the table names A and B. The parser creates nodes following a post-order traversal of matched grammar rules which emits children before their parents. This eliminates additional serialization steps since nodes can be written to the buffer while parsing. In the example, the relation names are matched as table references and form the two children of a `from`

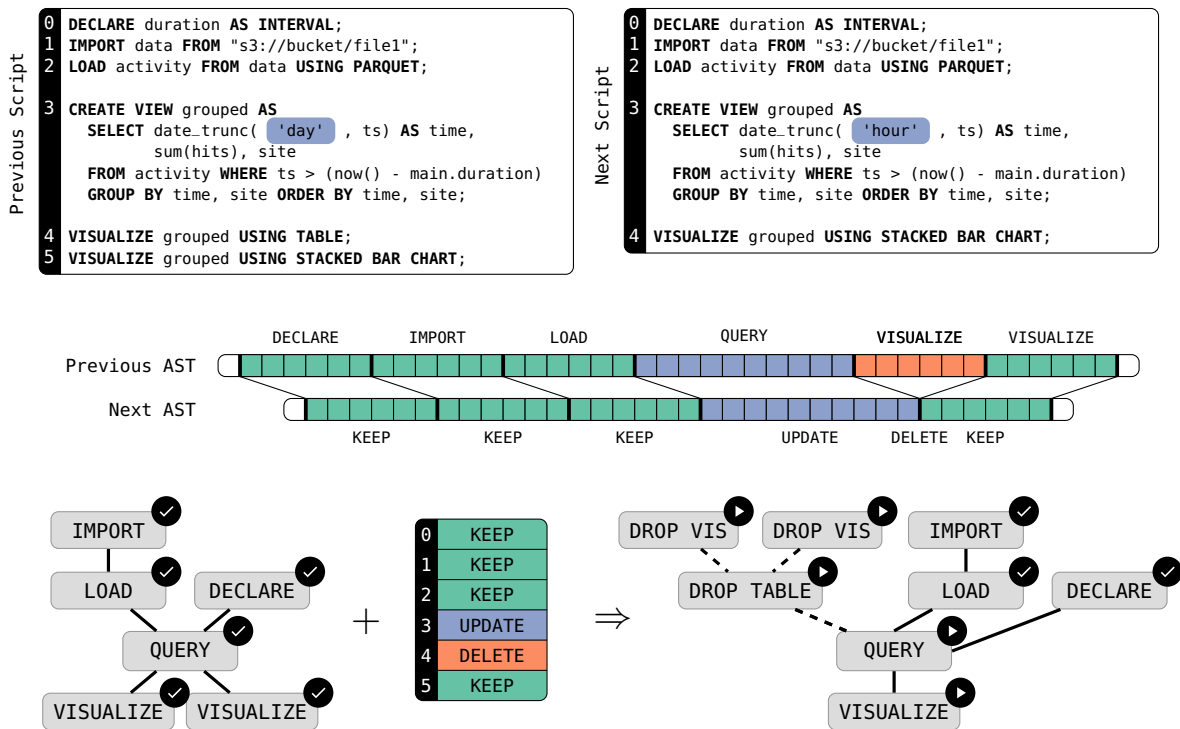


Figure 5: Example of a task graph that is derived from a previous task graph and an AST-based script difference. The two scripts visualize grouped timeseries data and differ in a deleted statement and the grouping granularity. The AST colors equal statements in green, changes in blue and deletions in orange.

clause. The output of the syntactic analysis is a program description representing statements as offsets of root nodes in this AST buffer. This representation is not only cache efficient but also simplifies the crossing of system boundaries as the consecutive memory buffer and fixed size nodes simplify the communication between system components and languages.

3.3.2 From AST to Task

The actionable units of our system are called tasks. Tasks are derived from statements and form a graph based on the statement dependencies. A query statement that references a LOAD statement, for example, translates into a query task that scans the output of a load task. These tasks are partially ordered and evaluate the entire script starting with data ingestion and ending with the visualization of derived tables. New tasks are derived from every user interaction based on the difference between the AST and its predecessor. This

includes tasks to undo the effects of deleted statements, update the effects of modified statements and add the effects of new statements.

For example, if a SQL statement that created a table is deleted from the script, the system derives a task to undo the effects by dropping the table. This mechanism is more abstract than the traditional transaction isolation of database systems as all tasks maintain a single workflow state that is updated with respect to changes in the script text and the user input. `VISUALIZE` statements, for example, compile Vega-Lite specifications only once and delete or update the specification only when the statement changes. `IMPORT` statements that download data using HTTP will further cache the data until a script change invalidates the output. The task graph drives the execution of an analysis workflow and serves as an anchor for any operations on the derived state.

3.3.3 Adaptive Task Graphs

The task graph of DashQL is adaptive as it reflects all continuous changes in the script and the user input. We implement a variant of an algorithm known as *Patience Diff* that is implemented in the version control systems *GNU Bazaar* and *Git*. The algorithm derives task updates from the difference between two scripts and works as follows:

We first determine all unique statement mappings between a script and its predecessor. Two statements are compared based on their ASTs instead of texts for whitespace insensitivity and support for incremental changes. The similarity can be quantified by counting equal AST nodes in two simultaneous DFS traversals and weighting them by the distance to the AST root. The tree traversals profit from the compact and cache-efficient encoding of nodes into a single AST buffer. Next, we compute the longest common subsequence among the mapped statements and use them as anchors for the remaining assignments. The remainder is then iterated in sequence and assigned to the most similar matches that have not been assigned yet. This identifies new and deleted statements and emits a similarity score for the rest.

Afterward, we determine the *applicability* of all previous tasks. A task is *applicable* if it was derived from a statement that stayed the same, does not *transitively* depend on an *inapplicable* task, and is not followed by an *inapplicable* task that successfully modified the own output. The *applicability* can be determined through a single DFS traversal with backward propagation when encountering an *inapplicable* task. *Applicable* tasks and their state are migrated

and marked as completed while the effects of all other tasks are updated or undone.

Figure 5 illustrates the entire process with an example of two scripts that analyze site activity data stored in AWS S3. The first script starts with a `DECLARE` statement that receives a time interval for the analysis. It then downloads the data from an AWS S3 bucket using a `IMPORT` statement and inserts the data into the database as a Parquet file using `LOAD`. The statements are followed by a traditional SQL query to filter the site activity in the input interval and compute aggregates grouped by days. The final two statements visualize the result of this query as a table and a stacked bar chart. The second script is almost identical to the first one except that the data is now grouped per hour instead of days and is no longer visualized as table.

AST buffers of both scripts are shown below, with the node color indicating the statement differences. The first three statements and the last are equal and therefore do not need to change. The query statement differs in the string literal that is passed to the function `date_trunc` and is marked as updated. The first visualization statement is no longer present in the new script and is marked as deleted.

The figure also contains a task graph derived from the previous script. It shows one task for every statement and a checkmark indicating that all of them were successfully executed. This task graph is then combined with the computed statement mappings to derive a new set of tasks reflecting the changes between the scripts. The table visualization was deleted, emitting a task called `DROP VIZ` to remove the table. The query statement was updated and results in the task `DROP TABLE` to undo the effects of the SQL query. However, this effect propagates since both visualizations depend on the table data. Therefore, we also undo the second visualization's effects and recreate it after executing the updated SQL statement. The remaining tasks that fetch and load the remote data into the database and receive input from the user are migrated and marked as completed.

This example demonstrates differences with the traditional script execution in relational database systems. DashQL defines entire analysis workflows, including external data, visualizations, and interactions with a user. Scripts are therefore not evaluated independently but in the context of a preceding execution, rewarding awareness of the existing state.

3.3.4 Complementing Vega-Lite

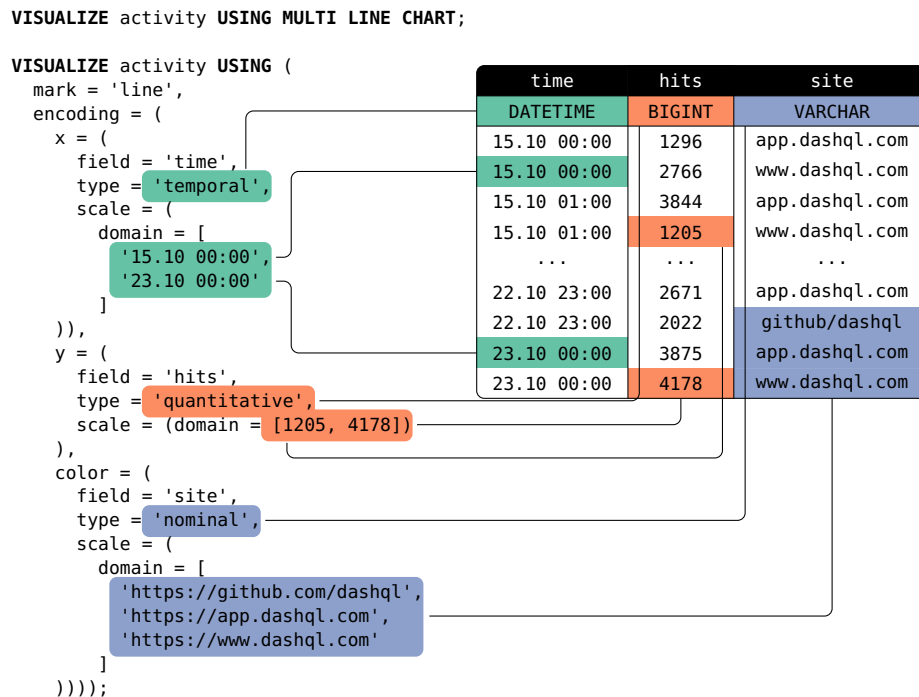


Figure 6: Two `VISUALIZE` statements that produce the same time series line chart, showing website hits of multiple websites. DashQL generates Vega-Lite specifications based on the table schema and statistics.

Vega-Lite offers a grammar to describe an expressive range of charts in declarative JSON specifications. The `VISUALIZE` statement of DashQL supports Vega-Lite specifications as nested key-value pair lists in SQL. `VISUALIZE` does not need to embed its own grammar of graphics, and users already familiar with Vega-Lite do not have to learn a new language.

Vega-Lite specifications are self-contained and describe visualizations without the context of an existing data model. In DashQL, visualizations are always backed by SQL queries which offer an opportunity to auto-complete parts of a specification. This reduces the pressure on Vega-Lite and pushes costly data introspection into the database system. Examples of this are encoding types and scale domains. We know the data types of all involved attributes based on the SQL metadata, which enables robust defaults, for example, when selecting between *quantitative*, *ordinal*, and *nominal* encoding types. Additionally, we can determine a value domain or range efficiently upfront using SQL queries.

DashQL further provides simplified `VISUALIZE` statements that can be written in tandem with SQL queries. This follows the observation that explicit defaults can guide the writing of SQL queries for subsequent visualization. For example, users can express a preferred field assignment through attribute aliases. A projection like `SELECT time AS x, hits AS y, site AS color ...` implicitly provides the fields for a visualization that can be reduced to `VISUALIZE ... USING MULTI LINE CHART`. These mappings can still be overruled by explicit settings but enable short statement variants for fast exploration. Users can default to listing attributes in a specific order without matching aliases. By default, DashQL assigns the first three attributes to the encoding channels `x`, `y`, `color`, resulting in the same output.

Figure 6 lists two `VISUALIZE` statements. The first one presents a simplified syntax that instructs DashQL to draw a chart with multiple lines. The second one describes the same output using an embedded Vega-Lite specification. Internally, DashQL uses the table on the right to derive the specification of the second statement for the first one. The table has the three columns `time`, `hits` and `site` with the data types `DATETIME`, `BIGINT` and `VARCHAR`. By default, a chart with multiple lines requires encoding declarations for `x`-values, `y`-values and the line `color`. Without further hints, DashQL assigns the columns in-order, using the `time` attribute for `x`, the number of `hits` for `y`, and the `site` name as `color`. The encoding types are derived based on the encoded column names and the column data types. An `x`-encoding backed by a `DATETIME` attribute is *temporal*, by default. The other encoding types are *quantitative* for `y`-values of type `BIGINT` and *nominal* for `color`-values of type `VARCHAR`. DashQL then resolves the domain for each of the scales in the encodings. The domains of temporal and quantitative scales are computed using minimum and maximum aggregates and yield the interval between the 15th and 23rd of August for the `time` attribute and the value range between 1205 and 4178 for the hit count. The domain of the nominal scale is then resolved by querying distinct `site` values, emitting the three websites of the DashQL project.

In summary, Vega-Lite provides a robust grammar for declarative visualizations in DashQL. We further extend the capabilities of Vega-Lite by completing specifications based on the contextual query metadata. This accelerates the data exploration without losing the flexibility of a complete specification whenever needed.

3.3.5 Language Extensions

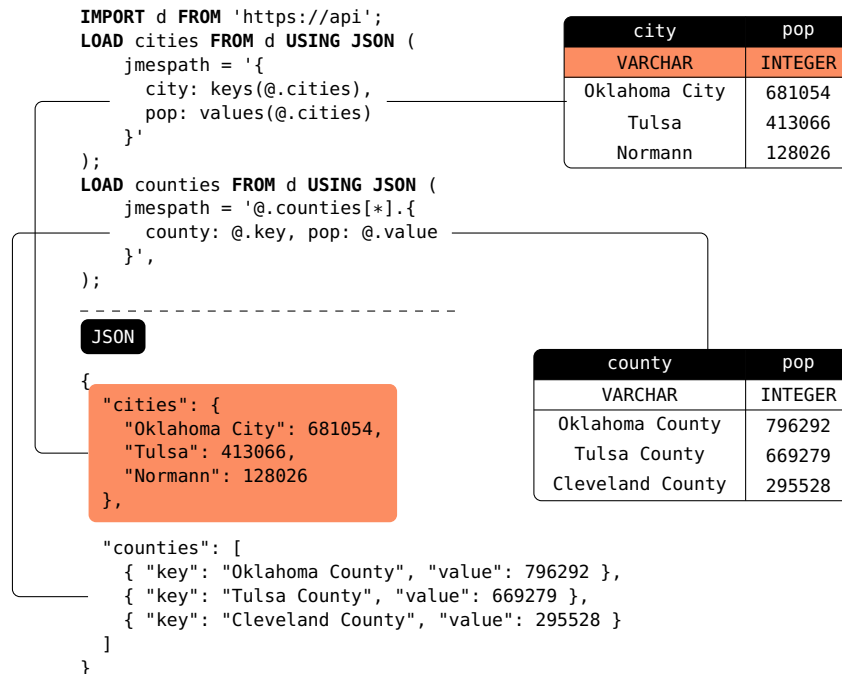


Figure 7: Two load statements that extract two relations from a single JSON document using JMESPath expressions. Both expressions extract populations in Oklahoma. The first expression emits the city data in column-major format, the second expression returns county data in row-major format.

The syntax of the DashQL statements `DECLARE`, `IMPORT`, `LOAD` and `VISUALIZE` end with optional settings provided as key-value pair lists. This offers a mechanism to extend DashQL without modifying the grammar rules or the language model. The settings translate to a generic dictionary passed to the derived tasks. Custom task implementations can read this dictionary and enable extensions based on available keys.

Our reference implementation, for example, extends the loading of JSON data through JMESPath expressions. Our embedded database DuckDB-Wasm can default load a table from a JSON document in two formats. Either in row-major format as a top-level array of objects where each object contains all attributes of the relation or in a column-major format as a top-level object with members storing column arrays. If a JSON document is not in either of those formats, it has to be transformed first. For this, the JSON task checks for the

key "jmespath" in the settings. If it is present, the task evaluates the expression on the input data first before loading it into the database.

Figure 7 lists an example DashQL script that loads two relations from a single JSON document that was returned from a remote HTTP API. The document stores population data of Oklahoma. City populations are stored as a single object with city names as properties, whereas county populations are provided as an array of objects. The first expression emits an object with the field *city* storing an array of city names and the field *pop* holding an array of population values. The second expression returns the county object array with changed attribute names. This example demonstrates the extensibility of the DashQL language through custom task implementations that can be configured through dynamic configuration options.

3.3.6 Holistic Optimization

Data transformations can be expensive, which makes their optimization indispensable for every data analysis workflow. Therefore, query optimizers are a vital component of every data processing system today and significantly impact overall execution times. Research around query optimization is profound and has been expanded for decades. Nevertheless, databases are universal and face the difficult task of accelerating specific queries without losing the generality. As a result, database systems rarely include external information during planning, leaving these non-trivial problems to the applications. DashQL unifies the data retrieval, transformations, and visualizations in the same language, which presents an opportunity for holistic optimizations.

Visualization-Driven Aggregation

The first example of holistic optimization is the automatic aggregation of SQL results for VISUALIZE statements. Jugel et al. introduced the value-preserving aggregation M₄ [54] to accelerate the visualization of time series data. M₄ follows the observation that the amount of rendered data points in line charts can be limited by the number of visible pixels on the screen. Instead of visualizing every single tuple of a time series, we can select a subset of the tuples based on the chart dimensions. The authors group values by time bins and compute the four name-giving aggregates $\min(x)$, $\max(x)$, $\min(y)$, and $\max(y)$ per bin. The associated points span a bounding box around all tuples in a bin that intersects any pixels that should be colored

```

WITH m4 AS (
  SELECT round($width * (x - $lb) / ($ub - $lb)) as k,
         min(x) AS min_x, max(x) AS max_x,
         min(y) AS min_y, max(y) AS max_y
  FROM $user_data GROUP BY k)
SELECT x, y FROM m4, $user_data
WHERE k = round($width * (x - $lb) / ($ub - $lb))
AND (y = min_y OR y = max_y OR
     x = min_x OR x = max_x)

```

Figure 8: M₄, a query for value-preserving time series aggregation, described by Jugel et al [54]. This version uses a CTE instead of a subquery with equal semantics.

for the line chart. With DashQL, introducing M₄ becomes an optimization that propagates the visualization context towards the backing SQL query.

M₄ is a value-preserving time series aggregation equivalent to the one listed in Figure 8. The query scans the relation `user_data` and computes the four aggregates grouped by a bin key. Afterward, the query resolves the corresponding `x`- and `y`-values of the aggregates by joining the aggregates again with the input data. A tuple in the input qualifies in that join, if an aggregate exists with the same key and either `x` or `y` equals an extreme value. The query does not rely on any specific aggregation functions, making it compatible with various database systems.

However, the original version of M₄ introduces a subtle but essential assumption. It scans the input relation twice and joins the extreme values to reconstruct the corresponding input tuples. This assumes that the extreme values are unique as the join might otherwise emit duplicates. For example, a constant function like $f(x) = 42$ will resolve 42 as minimum and maximum `y` value of every group. The following join will then emit the entire input relation since all tuples contain the same value for `y`. To support non-unique `y`-values, we therefore also have to make the output distinct on `k`, `x`, and `y`. As a result, M₄ consists of a repeated scan, a join, and two aggregations or otherwise has to fall back to significantly slower window functions.

We propose an alternative version of M₄, called AM₄, shown in Figure 9. It uses the aggregation functions `arg_min` and `arg_max`, sometimes implemented as `min_by`, and `max_by`, that are provided by several databases today (e.g., by ClickHouse, DuckDB and Presto). The function `arg_min(a, b)` selects an arbitrary attribute for `a` where `b` is minimal and can be computed alongside a

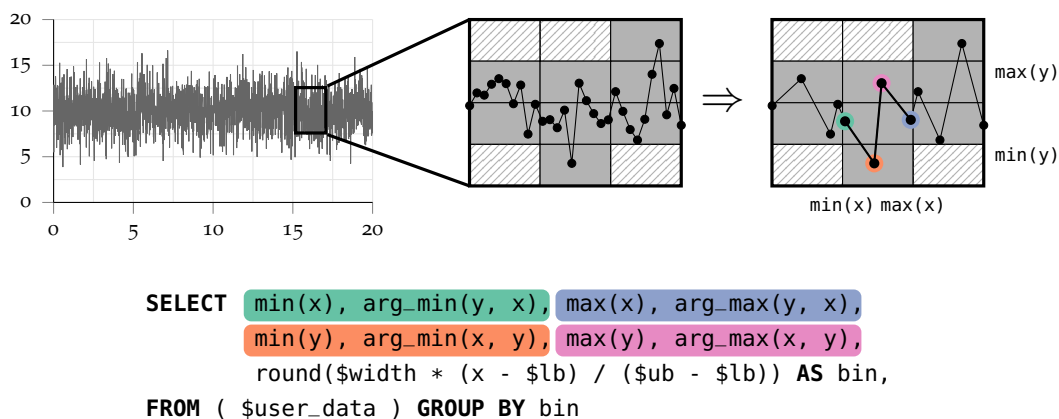


Figure 9: AM₄, a more efficient version of M₄ that provides value-preserving time series aggregation using a single scan and the aggregation functions `arg_min` and `arg_max`.

`min(b)` aggregate at negligible costs. We extend M₄ by additionally computing the aggregates `arg_min(y, x)`, `arg_max(y, x)`, `arg_min(x, y)` and `arg_max(x, y)`. This resolves existing points associated with the extreme values in a single efficient grouping, eliminating the second scan and the distinct aggregation.

Adaptive Materialization

The second example of holistic optimization is called Adaptive Materialization. DashQL statements like `IMPORT` and `LOAD` only *specify* data sources and formats. It is left to the optimizer to decide at runtime if the file contents should be materialized as a table upfront or if the data should be loaded lazily as part of the following SQL query. This decision depends not only on a single query but the entire script context, as multiple statements might refer to the same data. If the file format allows it, DashQL can use projection and predicate pushdown of databases only to fetch relevant parts of a file based on the specific query.

Predicate pushdown is a common optimization technique in databases and describes the evaluation of predicates as far down in the query plan as possible. The direction *down* refers to the widespread representation of relational algebra where relations form *leaves* of a tree that are combined using joins. When optimizing relational algebra, a common task is to push individual predicates towards these *leaves* to reduce the cardinality of relation as early as possible. If such a predicate is evaluated right after scanning file formats like Parquet, the

database can evaluate the predicates on file statistics and skip reading entire row groups.

For example, the database DuckDB supports reading remote Parquet files partially using an HTTP filesystem and skips row groups based on predicates in the table function `parquet_scan`. With DuckDB, DashQL fetches and loads the Parquet files in following SQL queries, if the data is not consumed by multiple statements. On the other hand, formats like CSV require downloading and parsing the entire file, independent of subsequent filters. In these cases, DashQL materializes the CSV contents once and shares the table with all the following statements. The decision to materialize data depends on the data source, the data format, all queries in the script, and the underlying database's capabilities. We call this technique Adaptive Materialization and see it as an opportunity to replace traditional caching logic with query-driven optimization passes.

3.4 EXAMPLE DATA EXPLORATION

We demonstrate data exploration with DashQL by constructing an example analysis workflow. The example analyzes a dataset with website activity data and builds a dashboard to view daily total page views for individual websites. We describe the textual changes to the script in every step and how they affect the reevaluation of the derived task graph. The script text and the associated output of the tool are shown in Figure 10.

Data Input. Our exploration begins with a declaration of the workflow's input data. The first script is labeled with ① and consists of three DashQL statements. A `IMPORT` statement declares that a file with name `data` can be retrieved using HTTP, a `LOAD` statement interprets this data as Parquet file and a `VISUALIZE` statement colored in `green` displays the file contents. The figure also presents the output of the first statements that visualizes the unaggregated site activity data using a single table. This table is virtualized, which means that only visible rows are rendered. In SQL, this virtualization translates to `LIMIT` and `OFFSET` clauses to only query the relevant subset of the data. With a coherent language model, we can propagate the `LIMIT` and `OFFSET` specifiers towards the data retrieval during an optimization pass. As a result, this first step only reads the file metadata and the first bytes of the Parquet file using HTTP range requests. When the user scrolls through the data, the table dynamically

reads following tuples by adjusting both specifiers. The internal WebAssembly database also uses an accelerating readahead buffer for the remote file to minimize the number of roundtrips to the remote server. This reduces the latency that users have to wait until seeing a visualization and provides a graceful fallback to large reads when the data is being requested.



Figure 10: Authoring an example analysis workflow with DashQL. The workflow explores website activity data in four steps. The steps are labeled with ① to ④ and associate textual changes in the script with adjusted visual output. Visualization statements are colored in green, the input statement and the corresponding predicate in orange.

Aggregate Views. Next, we want to aggregate the site activity to inspect the hourly sum of page views. We modify the script as shown in ② and add an explicit SQL statement that groups the site activity data as well as an ad-

ditional VISUALIZE statement in **green** to display the aggregates using an area chart. During reevaluation, the former workflow state is left untouched since the previous statements were neither modified, nor invalidated. The new query statement, however, needs to scan the attributes `timestamp` and `views` of all tuples in the Parquet file to compute the new aggregates. The additional visualization statement waits for the grouping to complete and then displays an area chart. This demonstrates the generation of Vega-Lite specifications as outlined in Section 3.3.4 since the tool automatically selects the time and sum attributes for the x - and y -values and identifies temporal and quantitative axes.

Filter Website. The next step makes the analysis dashboard interactive. Instead of showing the total page views across all websites, we want to filter the activity data by a website name that is provided dynamically by the user. For this, ③ introduces an DECLARE statement colored in **orange** and includes a filter predicate in the SQL statement. The new input with name `website` is of type `VARCHAR` and displays a text field on top of the previous area chart. The added filter predicate checks if the website is either `NULL` or if the website attribute of the tuple equals the website variable in the script. By default, the input value will be `NULL` which means that the dashboard will show the total page views until a website name is entered. During reevaluation, the Patience Diff algorithm identifies the additional `WHERE` clause in the query statement and marks it as updated. The system therefore drops and recreates the grouped activity table as well as the area chart that consumes its data. The query now filters the attribute `website`, which means that an additional column needs to be fetched from the remote Parquet file. This input statement shows the capability of DashQL to parameterize any SQL statement without explicit text instantiation. The AST allows us to reference the input variable by qualifying its name with the default schema.

Polish Aesthetics. The last step polishes the aesthetics of the generated analysis dashboard. The short syntax of DashQL offers a frictionless visualization of arbitrary SQL statements but may be insufficiently generic for a final workflow output. For example, the former area chart visualization falls back to the SQL attribute names for axis labels and default colors for the covered area. As described in Section 3.3.4, DashQL internally lowers the short syntax to verbose specifications. To adjust fine-granular settings, DashQL can therefore rewrite existing statements and specify all lowered options explicitly. ④ demonstrates this by replacing the single area chart visualization with explicit settings after interacting with the previously rendered chart. It uses the verbose specifica-

tion to adjust the title, the axis labels, the tick count and the area opacity in the workflow script.

This example demonstrates that DashQL allows for a progressive construction of analysis workflows. The interplay between textual adjustments and continuous visualizations provides short feedback loops during the data exploration. Propagating limit and offset specifiers is an example for a holistic optimization that reduces the amount of loaded data based on user input.

3.5 VISUALIZATION WITH AM4

In this section, we measure the performance of AM4, a visualization-driven aggregation and an example for a holistic optimization in DashQL. As described in Section 3.3.6, AM4 accelerates chart rendering and reduces the total amount of downloaded data in a client-server setting by filtering minimum and maximum values of grouped data. We want to demonstrate the effects of this optimization by analyzing render and download times with increasing data sizes. The experiments were performed on a Ryzen 5800X CPU with Node.js v17.6.0 that is powered by the V8 engine v9.6.

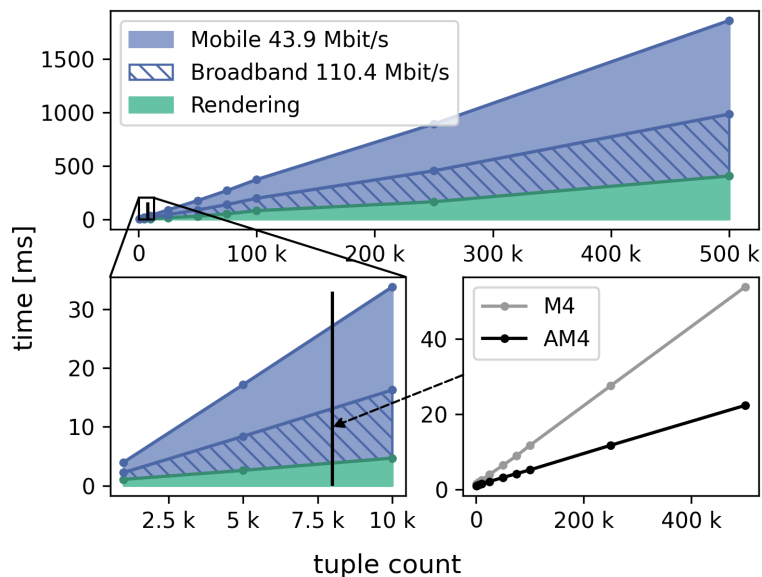


Figure 11: Downloading and rendering dominate the visualization times for increasing data sizes in a client-server setting. M4 and AM4 efficiently reduce large datasets to a small cardinality that can be visualized quickly.

Figure 11 contains three plots. The plot at the top shows in green color the time it takes to draw all points on a Cairo-backed canvas, using a prepared Vega view. It further adds the time to download the data in blue with either mobile (Cellular) or fixed broadband speeds. The Cisco Annual Internet Report [53] projects average global network performances of $110.4 \text{ Mbit s}^{-1}$ for fixed broadband and 43.9 Mbit s^{-1} for mobile networks by 2023. We assume a small record size of 16 B and compute the required time to download all tuples without any network latencies. Both durations present a significant delay especially for data sizes beyond 100k tuples that hamper any interactive exploration. The plot in the bottom right shows execution times of M₄ and AM₄. If we assume a canvas width of 1000 pixels and a device pixel ratio of 2, M₄ and AM₄ reduce the data cardinality to 8k tuples. AM₄ computes the aggregates for a relation with 500k entries in 22.3 ms and is twice as fast as M₄ which takes 53.7 ms. The plot in the bottom left shows the render and download and times for up to 10k tuples. A vertical line marks the resulting 8k tuples emitted by both algorithms that can be visualized quickly.

The experiment shows that both, M₄ and AM₄, accelerate the visualization of large data sets. This holds even when computing the analysis locally without downloads since rendering alone becomes expensive with an increasing number of tuples. AM₄ is therefore a good example for an optimization that propagates data from visualizations, such as the canvas width, back to the SQL query.

3.6 RELATED WORK

DashQL builds on ideas from declarative visualization and analysis languages and automatically optimizes workflows to make them more scalable.

3.6.1 Declarative Analysis Languages

Visualization and analysis languages are fundamental to exploratory analysis, either as a programming interface or as the underlying representation of a UI tool.

Declarative, textual languages provide a high-level notation to describe data science workflows. They often come with runtimes that optimize the data representation and query execution. These advantages make them a popular

choice over imperative languages like Python, R, or JavaScript. For example, SQL remains a popular tool for data scientists to express queries to databases decades after its invention [20]. Additionally, many declarative visualization and analysis languages have emerged. Vega [105] and Vega-Lite [104], for example, describe visualizations in JSON syntax. Their runtimes reduce redundant computation in these specifications and fill in rendering details. DashQL extends this research around declarative visualizations by integrating Vega-Lite specifications in the VISUALIZE statements. Vega also supports declarative data loading and transformations but authoring and debugging them can be cumbersome [51] and is often not performant enough. Dedicated analysis languages can fill this gap, for example by extracting analytical queries into explicit steps that can be annotated and tracked. Glinda [112] is a declarative format for specifying data science workflows including data loading, transformation, machine learning, and visualization. In contrast to Vega and Vega-Lite, Glinda describes analysis steps in YAML. DashQL follows the principles of declarative analysis languages but extends the language SQL instead. This approaches the goal of a coherent analysis language from the opposite direction as data ingestion and visualization are embedded into the database query language itself.

Vega, Glinda and DashQL, as most analysis languages, build on relational algebra and share a similar expressiveness in terms of the analyses they can describe [26]. Beyond the analysis steps, DashQL specifications describe inputs via UI widgets and outputs via tables and visualizations. Unlike Precision interfaces [139] and the recent PI2 [119] which implicitly generate UIs from SQL queries, the UI components in DashQL are explicitly described using the statements DECLARE and VISUALIZE. Like Vega visualizations, DashQL dashboards are interactive and update reactively to changes. Vega proposed a reactive runtime for visualizations [105] but all declarative components need to be specified by the author. When a declaration changes, the runtime needs to re-parse and re-evaluate the entire JSON specification.

When languages are used as model in a UI tool, analysts interactively modify an underlying specification that the system can reason about [48]. Polaris, which led to the creation of Tableau, explored this concept with the language VisQL [116]. In Voyager [131], people interactively change CompassQL [130] specifications and a recommender system suggests a gallery of visualizations. Lyra is an interactive visualization design environment that authors Vega-Lite specifications on behalf of the user [103]. We show in the chapter that DashQL

extends these ideas with an compact AST representation that allows for efficient updates.

Systems often blend code and graphical interfaces and allow modifications through either direct manipulation or code. Mage [58] and B2 [134] blur the boundaries between code and UI in Jupyter Notebooks. In Sketch-n-Sketch [50], people can write a program to generate graphics or manipulate the graphics directly in the rendering canvas. Inspired by these ideas, DashQL scripts describe visualizations like inputs, tables, and charts with text, but users can also change the statement by interacting with the UI. For example, DashQL offers to expand the short syntax of VISUALIZE statements or updates chart dimensions in the text when resizing the UI widget.

3.6.2 Scalable Visual Analysis

Even small latencies in visual analysis systems negatively affect people’s behavior during data exploration [78, 138]. Initial exposure to delays impair the subsequent performance even when delays are removed. Therefore, we want DashQL to respond to user interactions with low latency. DashQL builds on two ideas to achieve this goal.

First, DashQL uses an efficient in-browser analytical database based on DuckDB [100] that is further described in Chapter 6. The database allows to evaluate analysis workflows entirely on the client, avoiding costly roundtrips to a backend server. This lays the foundation for a distributed evaluation of workflows in the future that optimize dynamic client server scaling using a cost model [83]. Second, DashQL leverages the declarative format of analysis scripts to apply known optimizations from the database literature. These optimizations reduce redundant and unnecessary computations and avoid loading data that is not needed to answer a query. For example, DashQL reads data dynamically from remote files based on query predicates and projected attributes [10]. Propagating such information across statements shares similarities with provenance-supported interactions described by Psallidas et al. [98]. DashQL further implements a variant of the algorithm M4 [54] to reduce the rendering overhead with time series data.

Ideally, these optimizations happen transparently without the user having to manually specify them (as they would need to if they wrote their analysis in e.g., D3). Previous systems [79, 76, 84] used specialized engines to enable interactive response times. DashQL does not yet apply some of the indexing

techniques these systems proposed but it supports a wide range of analysis scenarios through general SQL queries. VegaPlus [136] is a related project that aims to improve performance of general visualizations by extracting data transformations from Vega [105] specifications and running them in a system that is more scalable than the Vega runtime. VegaPlus shifts computation but does not automatically apply data reduction techniques like M4.

3.7 SUMMARY

This chapter introduces the language DashQL. We list example scripts throughout the sections and discuss iterative data exploration in Section 3.4. The examples demonstrate the proximity of the language to SQL and the capability to describe complete analysis workflows. DashQL extends SQL by defining how data can be resolved and how results should be visualized. The coherent language model facilitates holistic optimizations covering data input, transforms and visualizations.

4

EVALUATING ADVANCED ANALYTICAL SQL QUERIES

Excerpts of this chapter have been published in [65].

4.1 INTRODUCTION

Users are rarely interested in wading through large query results when extracting knowledge from a database. Summarizing data using aggregation and statistics therefore lies at the core of analytical query processing. The most basic SQL constructs for this purpose are the associative aggregation functions SUM, COUNT, AVG, MIN, and MAX, which may be qualified by the DISTINCT keyword and have been standardized by SQL-92. Given the importance of aggregation for high-performance query processing, it is not surprising that several parallel in-memory implementations of basic aggregation have been proposed [101, 97, 71, 35, 85].

Chapter 2 lists additional aggregation functionality such as Window functions and grouping sets that joined associative aggregates in later versions of the standard. These constructs can also be used in conjunction, as the following SQL query illustrates:

```
WITH diffs AS (  
    SELECT a, b, c-lag(c) OVER (ORDER BY d) AS e  
    FROM R  
)  
SELECT a, b,                               -- window function (lag)  
       avg(e),                               -- associative aggregate  
       median(e)                             -- ordered-set aggregate  
       count(DISTINCT e), -- distinct aggregate  
FROM diffs  
GROUP BY (a, b)
```

The common table expression (WITH) computes the difference of each attribute *c* from its predecessor using the window function *lag*. For these differences, the query then computes the average, the median, and the number of distinct values.

Associative aggregates, ordered-set aggregates, and window functions not only have different syntax, but also different semantics and implementation strategies. For example, we usually prefer on-the-fly hash-based aggregation for associative aggregates but require full materialization and sorting for ordered-set aggregates and window functions. The traditional relational approach would therefore be to implement each of these operations as a separate relational operator. However, this has two major disadvantages. First, all implementations rely on similar algorithmic building blocks (such as materialization, partitioning, hashing, and sorting), which results in significant code duplication. Second, it is hard to exploit previously-materialized intermediate results. In the example query, the most efficient way to implement the counting of distinct differences may be to scan the sorted output of median rather than to create a separate hash table. An approach that computes each statistics operator separately may therefore not only require much code, but also be inefficient.

An alternative to multiple relational operators would be to implement all the statistics functionality within a single relational operator. This would mean that a large part of the query engine would be implemented in a single, complex, and large code fragment. Such an approach could theoretically avoid the code duplication and reuse problems, but we argue that it is too complex. Implementing a single efficient and scalable operator is a major undertaking [85, 73] – doing all at once seems practically impossible.

We instead propose to break up the SQL statistics functionality into several physical building blocks that are smaller than traditional relational algebra. Following Lohman [80], we call these building blocks *low-level plan operators (LOLEPOPs)*. A relational algebra operator represents a stream of tuples. A LOLEPOP, in contrast, may also represent materialized values with certain physical properties such as ordering. Like traditional operators, LOLEPOPs are composable – though LOLEPOPs often result in DAG-structured, rather than tree-structured, plans. LOLEPOPs keep the code modular and conceptually clean, while speeding up complex analytics queries with multiple expressions by exploiting physical properties of earlier computations. Another benefit of this approach is extensibility: adding a new complex statistic is straightforward.


In this chapter, we present the full life cycle of a query, from translation, over optimization, to execution: In Section 4.3, we first describe how to translate SQL queries with complex statistical expressions to a DAG of LOLEPOPs, and then discuss optimization opportunities of this representation. Section 4.4 de-











scribes how LOLEPOPs are implemented, including the data structures and algorithms involved. In terms of functionality, our implementation covers aggregation in all its flavors (associative, distinct, and ordered-set), window functions, and grouping sets. As a side effect, our approach also replaces the traditional sort and temp operators since statistics operators often require materializing and sorting the input data. Therefore, this chapter describes a large fraction of any query engine implementing modern SQL (the biggest exceptions are joins and set operations). We integrated the proposed approach into the high-performance compiling database system Umbra [88, 57] and compare its performance against HyPer [87]. We focus on the implementation of non-spilling LOLEPOP variants, and assume that the working-set fits into main-memory. The experimental results in Section 4.5 show that our system outperforms HyPer on complex statistical queries – even though HyPer has highly-optimized implementations for aggregation and window functions. We close with a discussion of related work in Section 4.6 and a summary of the chapter in Section 4.7.

4.2 BACKGROUND

Relational algebra operators are the prevalent representation of SQL queries. In relational systems, the life cycle of a query usually begins with the translation of the SQL text into a tree-shaped plan containing logical operators such as `SELECTION`, `JOIN`, or `GROUP BY`. These trees of logical operators are optimized and lowered to physical operators by specifying implementations and access methods. The physical operators then serve as the driver for query execution, for example through vectorized execution or code generation. System designs differ vastly in this last execution step but usually share a very similar notion of logical operators.

Database systems usually introduce at least two different operators to support the data analysis operations of the SQL standard. The first and arguably most prominent one, is the `GROUP BY` operator, which computes associative aggregates like `SUM`, `COUNT` and `MIN`. These aggregate functions are part of the SQL:1992 standard and already introduce a major hurdle for query engines in form of the optional `DISTINCT` qualifier. A hash-based `DISTINCT` implementation will effectively introduce an additional aggregation phase that precedes the actual aggregation to make the input unique for each group. When comput-

Table 1: LOLEPOPs for advanced SQL analytics. The input and output are either a tuple stream (, ) or tuple buffer (, ).

Operator	In	Out	Semantics	Implementation
Transform	PARTITION		Hash-partitions input	Materializes hash partitions (per thread), then merges across threads
	SORT		Sorts hash partitions	Sorts partitions with a morsel-driven variant of BlockQuicksort [33]
	MERGE		Merges hash partitions	Merges partitions with repeated 64-way merges
	COMBINE		Joins unique input on the group key	Builds partitioned hash tables after materializing input. Flushes missing groups to local hash partitions and then rehashes between pipelines
Compute	SCAN		Scans hash partitions	Scans materialized hash partitions and indirection vectors
	WINDOW		Aggregates windows	Evaluates multiple window frames for each row
	ORDAGG		Aggregates sort-based	Aggregates sorted key ranges. Scans twice for nested aggregates
	HASHAGG		Aggregates hash-based	Aggregates input in fixed-size local hash tables and flushes collisions to hash partitions, then merges partial aggregates with dynamic tables
*			Traditional operators	—

ing the aggregate `SUM(DISTINCT a) GROUP BY b`, for example, many systems are actually computing:

```
SELECT sum(a)
FROM (SELECT a, b FROM R GROUP BY a, b)
GROUP BY b
```

Now consider a query that contains the two distinct aggregates `SUM(DISTINCT a)`, `SUM(DISTINCT b)` as well as `SUM(c)`. If we resort to hashing to make the attributes `a` and `b` distinct, we will receive a DAG that performs five aggregations and joins all three aggregates into unique result groups afterwards. This introduces a fair amount of complexity hidden within a single operator.

Grouping sets increase the complexity of the `GROUP BY` operator further as the user can now explicitly specify multiple group keys. With grouping sets, the `GROUP BY` operator has to replicate the data flow of aggregates for every key that the user requests. An easy way out of this dilemma is the `UNION ALL` operator, which allows computing the aggregates independently. This reduces the added complexity but gives away the opportunity to share results when aggregates can be re-grouped. For example, we can compute an aggregate `SUM(c)` that is grouped by the grouping sets `(a, b)` and `(a)` using `UNION ALL` as follows:

```
SELECT a, b, sum(c) FROM R GROUP BY a, b
UNION ALL
SELECT a, NULL, sum(c) FROM R GROUP BY a
```

Order-sensitive aggregation functions like median are inherently incompatible with the previously-described hash-based aggregation. They have to be evaluated by first sorting materialized values that are hash-partitioned by the group key and then computing the aggregate on the key ranges. Sorting itself, however, can be significantly more expensive than hash-based aggregation which means that the database system cannot just always fall back to sort-based aggregation for all aggregates once an order-sensitive aggregate is present. The evaluation of multiple aggregates consequently involves multiple algorithms that, in the end, have to produce merged result groups. The optimal evaluation strategy heavily depends on the function composition which presents a herculean task for a monolithic `GROUP BY` operator.

The `WINDOW` operator is the second aggregation operator that databases have to support. Unlike `GROUP BY`, the `WINDOW` operator computes aggregates in the

context of individual input rows instead of the whole group. That makes a hash-based solution infeasible even for associative window aggregates. Instead, the WINDOW operator also hash-partitions the values, sorts them by partition and ordering keys, optionally builds a segment tree, and finally evaluates the aggregates for every row [73].

It is tempting to *outsource* the evaluation of order-sensitive aggregates to this second operator that already aggregates sorted values. Some database systems therefore rewrite ordered-set aggregates into a sort-based WINDOW operator followed by a hash-based GROUP BY. This reduces code duplication by delegating sort-based aggregations to a single operator but introduces unnecessary hash aggregations to produce unique result groups. For example, the median of attribute *a* grouped by *b* can be evaluated with a pseudo aggregation function ANY that selects an arbitrary group element:

```
SELECT b, any(v)
FROM (SELECT b, median(a) OVER (PARTITION BY b) AS v FROM R)
GROUP BY b
```

We see this as an indicator that relational algebra is simply too coarse-grained for advanced analytics since it favors monolithic aggregation operators that have to shoulder too much complexity. We further believe that this is an ingrained limitation of relational algebra rooting in its reliance on (multi-)set semantics and its inability to share materialized state between operators.

4.3 FROM SQL TO LOLEPOPS

In this section, we first introduce the set of LOLEPOPs for advanced SQL analytics and describe how to derive them from SQL queries. We then outline selected properties of complex aggregates and how LOLEPOPs can evaluate them efficiently.

4.3.1 LOLEPOPs

The execution of SQL queries traditionally centers around unordered tuple streams. Relational algebra operators like joins are defined on unordered sets which allows execution engines to evaluate queries without fully materializing both inputs. State-of-the-art systems evaluate operators by processing the

input tuple-by-tuple regardless of whether the execution engine implements the Volcano-style iterator model, vectorized interpretation or code generation using the push model. An exception to this rule are systems that always materialize the entire output of an operator before proceeding. This adds flexibility when manipulating the same tuples across several operators but the inherent costs of materializing all intermediate results by default is usually undesirable.

Our LOLEPOPs bridge the gap between traditional stream-based query engines and full materialization by defining operators on both unordered tuple streams and buffers. These buffers are further specified with the physical properties *partitioning* and *ordering* which allows reusing materialized tuples wherever possible. Table 1 lists eight LOLEPOPs that are both necessary and sufficient to compose advanced SQL aggregates. Of these, five *transform* materialized values and three *compute* the actual aggregates. The transform LOLEPOPs can be thought of as utility operators that prepare the input for the compute LOLEPOPs. For example, before one can compute an ordered aggregate using ORDAGG, the input data has to be partitioned and sorted. Buffers can be scanned multiple times, which allows decomposing complex SQL analytics into consecutive aggregations that pass materialized state between them.

For every LOLEPOP, Table 1 lists whether it produces and consumes tuple streams (▶, ◀) or tuple buffers (▣, ▢). These input and output types form the interface of a LOLEPOP and may be asymmetric. For example, the PARTITION operator consumes an unordered stream of tuples (◀) and produces a buffer that is partitioned (▣). The SORT operator, on the other hand, reorders elements in place and therefore defines input and output as buffer (▢ ▣). Together, the two operators form a reusable building block (◀ ▣) that materializes input values and prepares them, e.g., for ordered-set or windowed aggregation. This allows implementing complex tasks like parallel partitioned sorting in a single code fragment and enables more powerful optimizations on composed aggregates.

Most of the LOLEPOPs consume data from a single producer and provide data for arbitrary many consumers. The only exception is the operator COMBINE that joins multiple tuple streams on group keys. The operator differs from a traditional join in a detail that is specific to aggregation. It leverages that groups are produced at most once by every producer to simplify the join on parallel and partitioned input. We deliberately use the term COMBINE distinguishing the join of unique groups from generic sets.

We use dedicated LOLEPOPs to differentiate between hash-based (HASHAGG) and sort-based (ORDAGG) aggregation. Many systems implement these two fla-

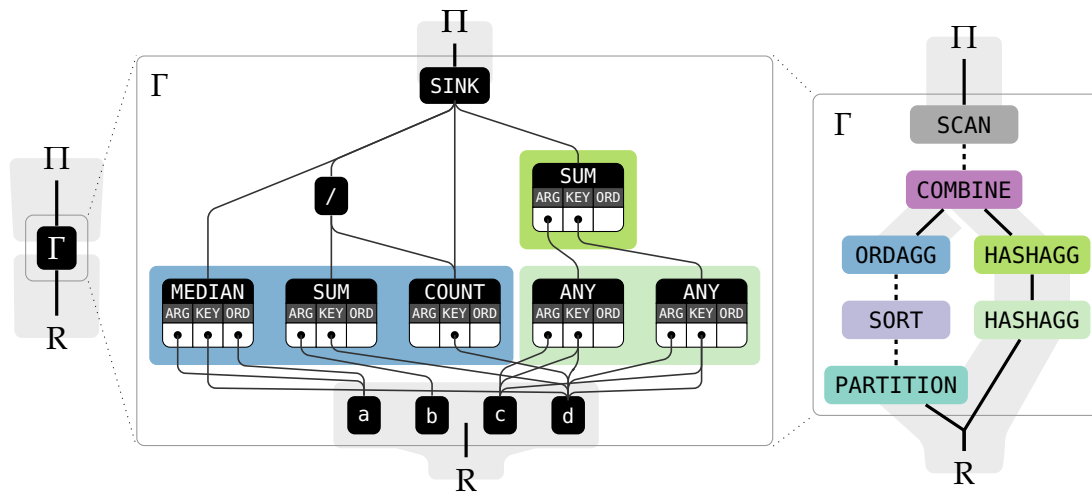


Figure 12: Translation of a GROUP BY operator into a computation graph to construct a DAG of LOLEPOPs.

SQL: `SELECT median(a), avg(b), sum(DISTINCT c) FROM R GROUP BY d`

vors of aggregation by lowering a logical GROUP BY operator to two physical ones. However, this choice is very coarse-grained since the result is still a single physical operator that has to evaluate very different aggregates at once. With our framework, database systems can freely combine arbitrary flavors of aggregation algorithms as long as they can be defined as a LOLEPOP. Section 4.3.3 discusses several examples that unveil the hitherto dormant potential of such hybrid strategies. For example, while associative aggregates usually favor hash-based aggregation, we may switch to sort-aggregation in the presence of an additional ordered-set aggregate. If the required ordering is incompatible, however, it may be more efficient to combine both, hash-based and sort-based aggregation.

4.3.2 From Tree to DAG

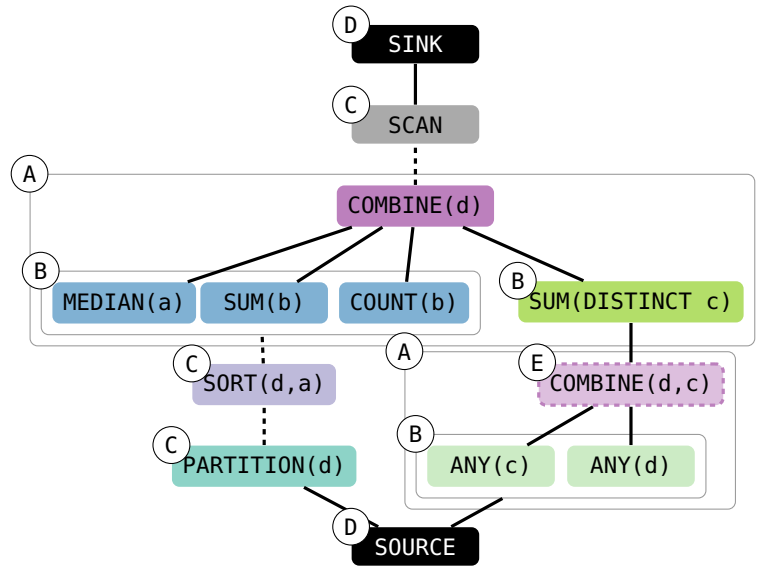
We derive the LOLEPOP plan from a computation graph that connects the input values, the aggregate computations as well as virtual source and sink nodes based on dependencies between them. Figure 12 shows, from left to right, a relational algebra tree, the derived computation graph, and the constructed DAG containing the LOLEPOPs for a query that computes the three aggregates median, average and distinct sum.

The relational algebra tree only consists of a scan, a monolithic `GROUP BY`, and a projection. At first, the `GROUP BY` aggregates are split up to unveil inherent dependencies of the different aggregation functions. The average aggregate, for example, is decomposed into the two aggregates `SUM` and `COUNT`, and a division expression. The distinct `SUM`, on the other hand, is first translated into `ANY` aggregates for arguments and keys followed by a `SUM` aggregate. `ANY` is an implementation detail that is not part of the SQL standard. It is a pseudo aggregation function that preserves an arbitrary value within a group and allows, for example, to distinguish the group keys from the unaggregated input values. Here, the attributes `c` and `d` are aggregated with `ANY`, grouped by `c, d` to make them unique.

The resulting aggregates and expressions are connected in the computation graph based on dependencies between them. A node in this graph depends on other nodes if they are referenced as either argument, ordering constraint or partitioning/grouping key. For example, the median computation references the attributes `a` and `d` whereas the any aggregates only depend on the attributes `c` and `d`. LOLEPOPs offer the option to compute the median independently of the distinct sum and then join the groups afterwards using the `COMBINE` operator. This results in DAG structured plans and requires special bookkeeping of the dependencies during the translation.

The computation graph of the example is translated into the seven LOLEPOPs on the right-hand side of the figure. The non-distinct aggregates in blue color are translated into the operators `PARTITION`, `SORT`, and `ORDAGG` since the median aggregate requires materializing and sorting all values anyway. The `ANY` aggregates, however, differ in the group keys and are therefore inlined as `HASHAGG` operator into the input pipeline. The distinct sum is computed in a second `HASHAGG` operator and is then joined with the non-distinct aggregates using the `COMBINE` operator.

The algorithm that derives these LOLEPOPs from the given computation graph is outlined in Figure 13. It consists of five steps that canonically map the aggregates to the LOLEPOP counterparts and then optimize the resulting DAG. Step ① collects sets of computations with similar group keys and constructs a single `COMBINE` operator for each set respectively. These `COMBINE` operators implicitly join aggregates with the same group keys and will be optimized out later if they turn out to be redundant. Step ② then constructs aggregate LOLEPOPs for computations within these sets. If the query contains grouping sets, it decomposes them into aggregations with separate grouping



- Ⓐ Add combine operators
- Ⓑ Compute aggregates
 - Expand grouping sets
 - Select aggregation order
 - Select aggregation strategies
- Ⓒ Propagate buffers
 - Add sorting operators
 - Add partitioning operators
 - Add scan operators
- Ⓓ Connect DAG
 - Consume from source operator
 - Produce for sink operator
- Ⓔ Optimize DAG
 - Replace unbounded windows
 - Remove redundant combines
 - Select producer order
 - Select buffer layouts
 - Select sort modes

Figure 13: Algorithm to derive the LOLEPOP DAG.

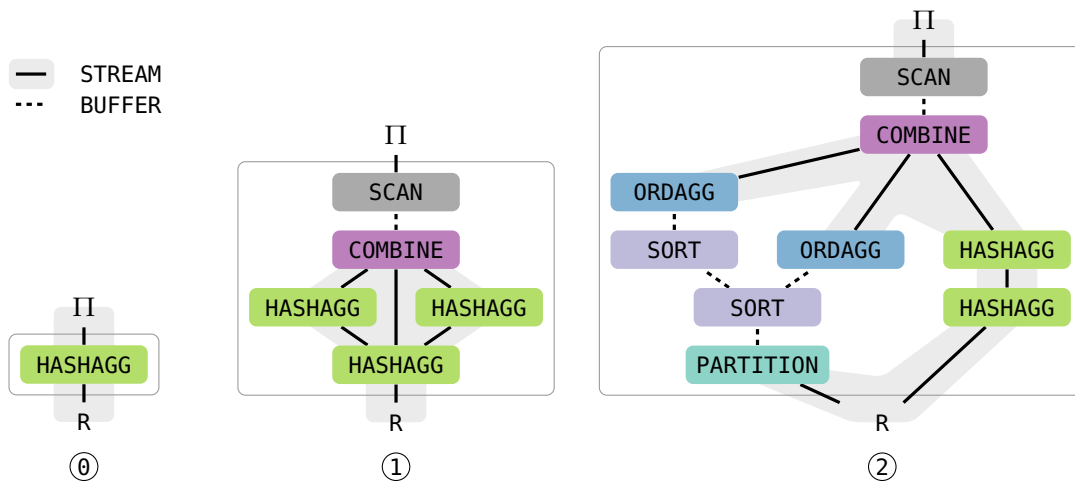
keys and adds them to the other aggregates attached to the combine operator. Afterwards, it divides the aggregates based on the grouping keys of their input values and determines favorable execution orders. In the example query, the operator `COMBINE(d)` joins the aggregates `MEDIAN(a)`, `SUM(b)`, `COUNT(b)` and `SUM(DISTINCT c)`. The first three aggregates depend on values that originate directly from the source operator whereas `SUM(DISTINCT c)` depends on values that are grouped by `d`, `c`. Among the first three, the aggregates `SUM` and `COUNT` are associative aggregates that would favor a hash-based aggregation. The `MEDIAN` aggregate, however, is an ordered-set aggregate and requires the input to be at least partitioned. The algorithm therefore constructs a single `ORDAGG` operator to compute the first three aggregates and a `HASHAGG` operator to compute the distinct sum. Step Ⓒ introduces all transforming operators

to create, manipulate and scan buffers. In the example query, this introduces the `SORT` and `PARTITION` operators required for `ORDAGG` as well as the final `SCAN` operator to forward all aggregates to the sink. Step ④ connects the source and sink operators and emits a first valid DAG.

Step ⑤ transforms this DAG with several optimization passes. The goal of these optimization passes is to detect constellations that can be optimized and to transform the graph accordingly. In the given query, the operator `COMBINE(d, c)` can be removed since there is only a single inbound `HASHAGG` operator. Other optimizations include, for example, the merging of unbounded `WINDOW` frames into following `ORDAGG` operators if the explicit materialization of an aggregate is unnecessary or the elimination of `SORT` operators if the ordering is a prefix of an existing ordering. In addition to graph transformations, these passes are also used to configure individual operators with respect to the graph. An example is the order in which `COMBINE` operators call their producers. If, for example, a `COMBINE` operator joins two ordered-set aggregates with different ordering constraints it is usually favorable to produce the operator "closer" to the source first to enable in-place reordering of the buffer. In general, such a favorable producer order can be determined with a single pre-order DFS traversal starting from the plan source. Another example, is the selection of the sorting strategy in the `SORT` operator and the propagation of the access method to consuming operators. Very large tuples may, for example, favor indirect sorting over in-place sorting which has to be propagated to a consuming `ORDAGG` operator.

The result is a plan of LOLEPOPs that eliminates many of the performance pitfalls that monolithic aggregation operators will run into. We do not claim that we always find the optimal plans for the given aggregations but instead make sure that certain performance opportunities are seized. We want to explore this plan search space using a physical cost model in future research.

The algorithm translates simple standalone aggregates into chains of LOLEPOPs. An associative aggregate with distinct qualifier, for example, is translated into the sequence `HASHAGG(HASHAGG(R))`. An ordered-set aggregate is computed on sorted input using `ORDAGG(SORT(PARTITION(R)))`. For a window function, we just need to replace the last operator and evaluate `WINDOW(SORT(PARTITION(R)))` instead. This already hints at the potential code reuse between the different aggregation types but does not yet take full advantage of DAG-structured plans.



- ① **SELECT a, var_pop(b), count(b), sum(b) FROM R GROUP BY a**
- ① **SELECT a, b, sum(c) FROM R GROUP BY GROUPING SETS ((a), (b), (a, b));**
- ② **SELECT a, sum(b), sum(DISTINCT b),
percentile_disc(0.5) WITHIN GROUP (ORDER BY c),
percentile_disc(0.5) WITHIN GROUP (ORDER BY d) FROM R GROUP BY a**

Figure 14: Plans for three example queries outlining challenges with composed aggregates, implicit joins and order sensitivity.

4.3.3 Advanced Expressions

Advanced expressions demand complex evaluation strategies. Figure 14 and Figure 15 show six example queries and the low-level plan.

Composed Aggregates must be split up to eliminate redundancy. The SQL standard describes various aggregation functions that can be decomposed into smaller ones. The aggregation function VAR_POP, for example, is defined as

$$Var(x) = \frac{1}{N} \cdot \sum_{i=0}^N (x_i - \bar{x})^2 = \left(\frac{1}{N} \cdot \sum_{i=0}^N x_i^2 \right) - \left(\frac{1}{N} \cdot \sum_{i=0}^N x_i \right)^2$$

and can be decomposed into

$$\frac{SUM(x^2) - \frac{SUM(x)^2}{COUNT(x)}}{COUNT(x)}.$$

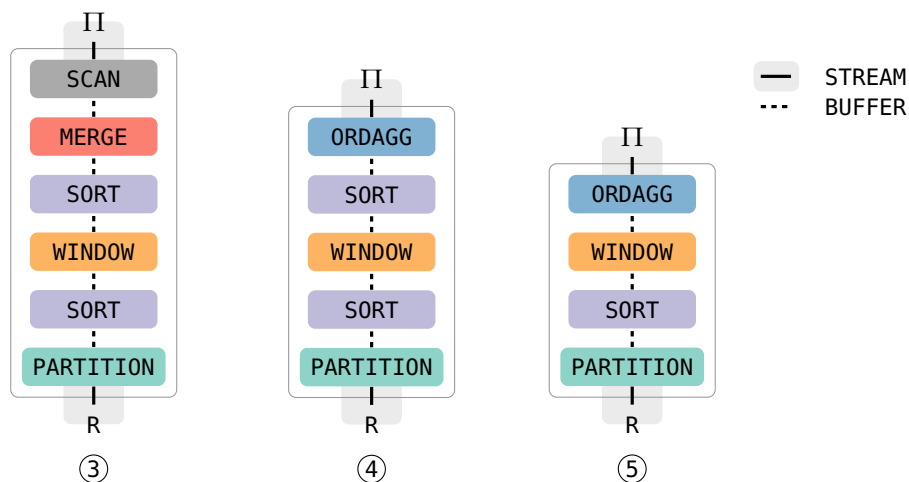
We have to share the aggregate computations among and within composed aggregates, which favors a graph-like representation of aggregates and expressions. © shows a query that computes the aggregates VAR_POP(x), SUM(x), and COUNT(x). We can evaluate all three aggregates with a single hash-based aggrega-

gation operator, but still have to infer that $SUM(x)$ and $count(x)$ can be shared with the variance computation.

Implicit joins are necessary whenever different groups need to be served at once. This can be the result of multiple order-sensitive and distinct aggregates or an explicit grouping operation such as `GROUPING SETS`. ① shows a query where an associative aggregate $SUM(c)$ is computed for the grouping sets (a) , (b) , and (a,b) . We can evaluate the query efficiently by inlining the grouping of (a,b) into the input pipeline and then grouping (a,b) by (a) and (b) . Afterwards, the output of all three aggregates is joined by (a,b) within a single hash-table. Grouping operations usually emit complicated graphs of LOLEPOPs and cause non-trivial reasoning about the evaluation order.

Order sensitivity has an invasive effect on the desirable plans since it usually requires materializing and sorting the entire input. This renders additional hash-based aggregation, which is often superior for standalone aggregation, inferior to aggregating sorted key ranges. ② shows a query that computes two order-sensitive aggregates $MEDIAN(c)$ and $MEDIAN(d)$ as well as two associative aggregates $SUM(b)$ and $SUM(DISTINCT b)$. One would usually prefer hash-bashed aggregation for the non-distinct sum aggregate to exploit the associativity. In the presence of the median aggregates, however, it is possible to compute the non-distinct sum on the same ordered key range and thus eliminate an additional hash table. The second median reuses the buffer of the first median and reorders the materialized tuples by (a,d) . The distinct qualifier leaves us with the choice to either introduce two hash aggregations, grouped by (a,c) and (a) , or to reorder the key ranges again by (a,c) and skip duplicates in `ORDAGG`. In this particular query, we use hash aggregations since the runtime is dominated by linear scans over the data as opposed to $\mathcal{O}(n \log n)$ costs for sorting. If the key range was already sorted by (a,c) , a duplicate-sensitive `ORDAGG` would be preferable.

Result ordering is specified through the SQL keywords `ORDER BY`, `LIMIT` and `OFFSET` and is crucial to reduce the cardinality of the result sets. We already rely on ordered input in the `WINDOW` and `ORDAGG` operators, which makes standalone sorting of values a byproduct of our framework. There are only two adjustments necessary. First, we have to support the propagation of `LIMIT` and `OFFSET` constraints through the DAG of LOLEPOPs to stop sorting eagerly. This can be implemented as pass through the DAG very similar to traditional optimizations of relational algebra operators. Additionally, we need a dedicated operator `MERGE` that uses repeated k -way merges to reduce the partition count efficiently.



- ③ `SELECT row_number() OVER (PARTITION BY a ORDER BY b) FROM R ORDER BY c LIMIT 10`
 ④ `SELECT a, mad() WITHIN GROUP (ORDER BY b) FROM R GROUP BY a`
 ⑤ `SELECT b, sum(pow(next_a - a, 2)) / nullif(count(*) - 1, 0)`
`FROM (SELECT b, a, lead(a) OVER (PARTITION BY b ORDER BY a) AS next_a`
`FROM R) GROUP BY b`

Figure 15: Plans for three example queries that allow for the optimization of result ordering and aggregate nesting.

③ shows a query that computes the window function `row_number` of attribute `b` and then sorts the results by an attribute `c`. The traditional approach would involve a dedicated operator on top that materializes and sorts the scanned output of the window aggregation. We instead just reorder the already materialized tuples by the new order constraint and eliminate the additional materialization.

Nested aggregates blur the boundary between grouped and windowed aggregations. The *Median Absolute Deviation* (MAD), for example, is a common measure for dispersion in descriptive statistics and is defined for a set $\{x_1, x_2, \dots, x_n\}$ as $MEDIAN(|x_i - \tilde{x}|)$ with $\tilde{x} = MEDIAN(x)$. \tilde{x} represents a window aggregate since $x_i - \tilde{x}$ has to be evaluated for every row. The outer median, however, is an order-sensitive grouping aggregation that reduces each group to a single value. One would like to try to transform this expression into a simpler form that eliminates the nested window aggregation, similar to the aforementioned variance function. However, the nature of the median prevents these efforts and one is forced to explicitly (re-)aggregate $x_i - \tilde{x}$. ④ shows a query that computes this MAD function. Our framework allows us to first compute the window aggregate and then reorder the key ranges for a following ORDAGG operator. This shows the power of our unified framework,

which blurs the boundary between the GROUP BY and WINDOW operators and can reuse the materialized output of $x_i - \bar{x}$.

Nested aggregates can also be provided by the user if the database system accepts window aggregates as input arguments for aggregation functions. The *Mean Square Successive Difference* (MSSD) is defined as

$$\sqrt{\frac{\sum_{i=0}^{N-1} (x_{i+1} - x_i)^2}{n - 1}}.$$

It estimates the standard deviation without temporal dispersion. ⑤ shows a query that computes the MSSD function by nesting the window aggregate LEAD into a SUM aggregate. A typical implementation would translate this query into a WINDOW operator followed by a GROUP BY. This, however, disregards the fact that the nested WINDOW ordering is compatible with the outer group keys. We can instead just aggregate the existing key ranges without further reordering using the ORDAGG operator.

4.3.4 Extensibility

We already mentioned a number of useful and widely-used statistics that are not part of the SQL standard, and many more exist. One advantage of our approach is that the computation graph facilitates the quick composition of new aggregation functions. We construct the computation graph using a planner API that lets us define nodes with attached ordering and key properties. We then use this API in Low-Level-Functions to compose complex aggregates through a sequence of API calls. In fact, we even implement the aggregation functions defined in the SQL standard as such Low-Level-Functions. The following example code defines the aforementioned *Mean Square Successive Difference* aggregate without explicitly implementing it in the operator logic:

```
def planMSSD(arg, key, ord):
    f = WindowFrame(Rows, CurrentRow, Following(1))
    lead = plan(LEAD, arg, key, ord, f)
    ssd = plan(power(sub(lead, arg), 2))
    sum = plan(SUM, ssd, key)
    cnt = plan(COUNT, ssd, key)
    res = plan(div(sum, nullif(sub(cnt, 1), 0)))
    return res
```

Other complex statistical functions like interquartile range, kurtosis, or central moment can be implemented similarly. Furthermore, a database system

can expose this API through user-defined aggregation functions. This allows users to combine arbitrary expressions and aggregations without the explicit boundaries between the former relational algebra operators.

4.4 LOLEPOP IMPLEMENTATION

In this section, we describe how the framework affects the code generation in our database system Umbra. We further introduce the data structures used to efficiently pass materialized values and outline the implementation of the LOLEPOPs `PARTITION` and `COMBINE` as well as the `ORDAGG`, `HASHAGG`, and `WINDOW` operators.

4.4.1 Code Generation

Umbra follows the producer/consumer model to generate efficient code for relational algebra plans [88, 57, 87]. In this model, operator pipelines are merged into compact loops to keep SQL values in CPU registers as long as possible. More specifically, code is generated by traversing the relational algebra tree in a depth-first fashion. By implementing the function `produce`, an operator can be instructed to recursively emit code for all child operators. The function `consume` is then used in the inverse direction to inline code of the parent operator into the loop of the child. Operators are said to *launch* pipelines by generating compact loops with inlined code of the parent operators and *break* pipelines by materializing values as necessary.

Figure 16 illustrates the code generation for a query that first joins two relations A and B and then computes the aggregates median, average, and distinct sum. The coloring indicates which line in the pseudocode was generated by which operator. On the left-hand side of the figure, the scan of the base relation B is colored in red and only generates the outermost loop of the second pipeline. The join is colored in orange and inlines code building a hash table into the first pipeline as well as code probing the hash table into the second pipeline. Both operators integrate seamlessly into the producer/consumer model since the generated loops closely match the unordered (multi-)sets at the algebra level. The group by operator, on the other hand, bypasses the model almost entirely. The code in yellow color partitions and sorts all values for the median and average aggregates and additionally computes the distinct

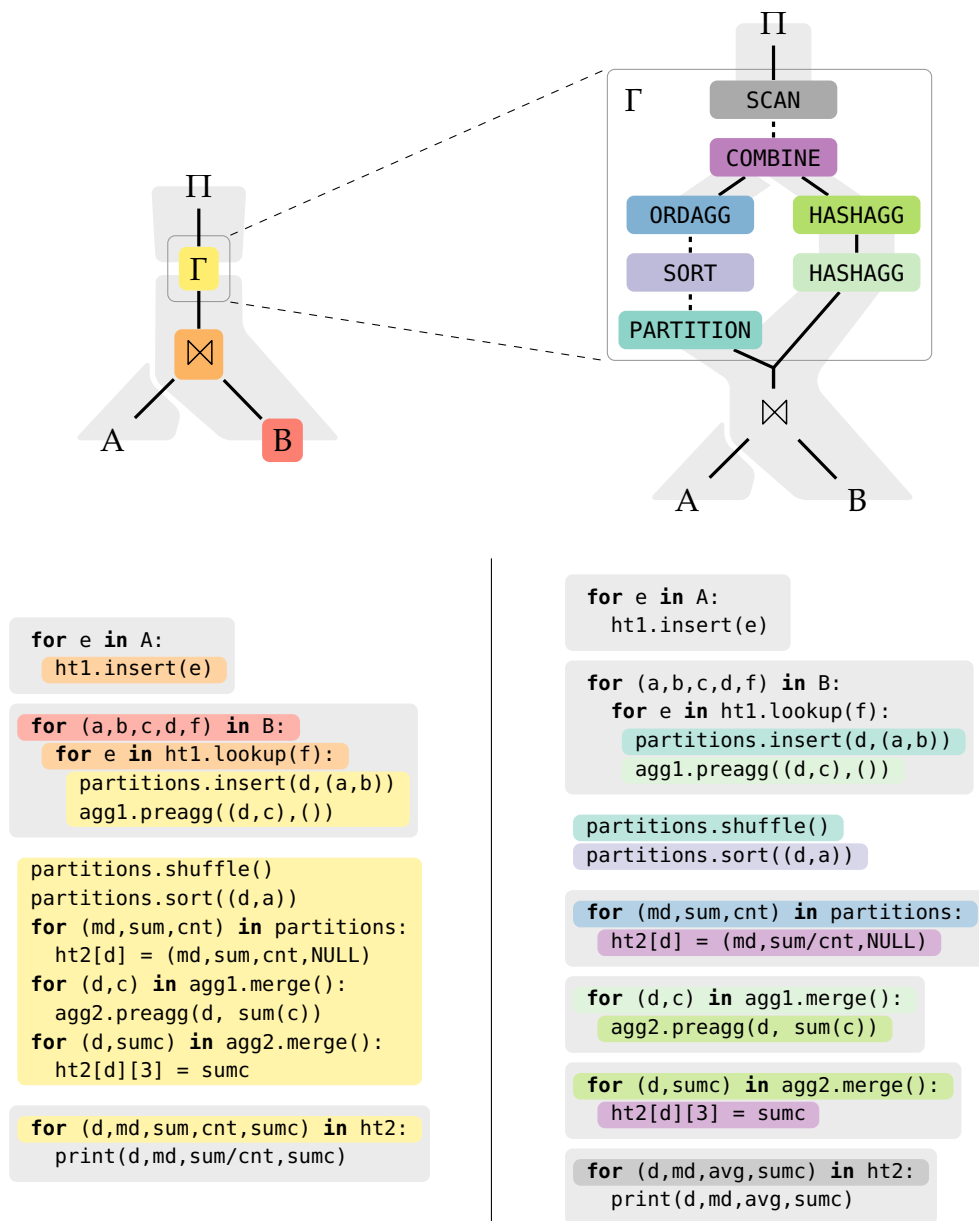


Figure 16: Plans and simplified code for a query that computes a median, an average, and a distinct sum of two joined relations.

sum via two hash aggregations. In contrast to the scan and join operators, most of this code is generated *in between* the unordered input and output pipelines since the aggregation logic primarily manipulates materialized values.

Our framework breaks this monolithic aggregation logic into a DAG of LOLEPOPs. Within the producer/consumer model, a LOLEPOP behaves just like every other operator with the single exception that it does not necessarily

call consume on the parent operator. Instead, multiple LOLEPOPs can manipulate the same tuple buffer via code generated in the produce calls.

These derived DAG that roots in the two outgoing edges of the former join operator. The producer/consumer model supports DAGs since we can inline both consumers of the join (the `PARTITION` and `HASHAGG` operators) into the loop of the input pipeline. The only adjustment necessary is to substitute the total order among pipeline operators with a partial order modeling the DAG structure. Our framework further unveils pipelines that have been hidden within the monolithic translation code. The output of the `ORDAGG` and the two `HASHAGG` operators represent unordered sets that can now be defined as pipelines explicitly. The dashed edges between the operators indicate passed tuple buffers as opposed to the solid edges for pipelines. In this example, data is passed implicitly between the operators `PARTITION`, `SORT`, and `ORDAGG` through the variable `partitions`. This lifts the usual limitation to only pass tuples in generated loops and allows us to compose buffer modifications.

In the example, the code that is generated through LOLEPOPs equals the code generated by the monolithic aggregation operator. This underlines that LOLEPOPs are not fundamentally new ways to evaluate aggregates but serve as more fine-granular representation that better matches the modular nature of these functions.

4.4.2 Tuple Buffer

The tuple buffer is a central data structure that is passed between multiple LOLEPOPs and thereby allows reusing intermediate results. Our tuple buffer design is driven by the characteristics of code generation as well as the operations that we want to support. First, the code generated by the producer-consumer model ingests data into the buffer on a tuple-by-tuple fashion. We also do not want to rely on cardinality estimates, which are known to be inaccurate [72]. Yet, we want to avoid relocating materialized tuples whenever possible. This favors a simple list of data chunks with exponentially growing chunk sizes as the primary way to represent a buffer partition. Second, we prefer a row-major storage layout for the tuple buffer. Our system implements a column-store for relations but materializes intermediate results as rows to simplify the generated attribute access. This will particularly benefit the `SORT` operator since it makes in-place sorting more cache-efficient.

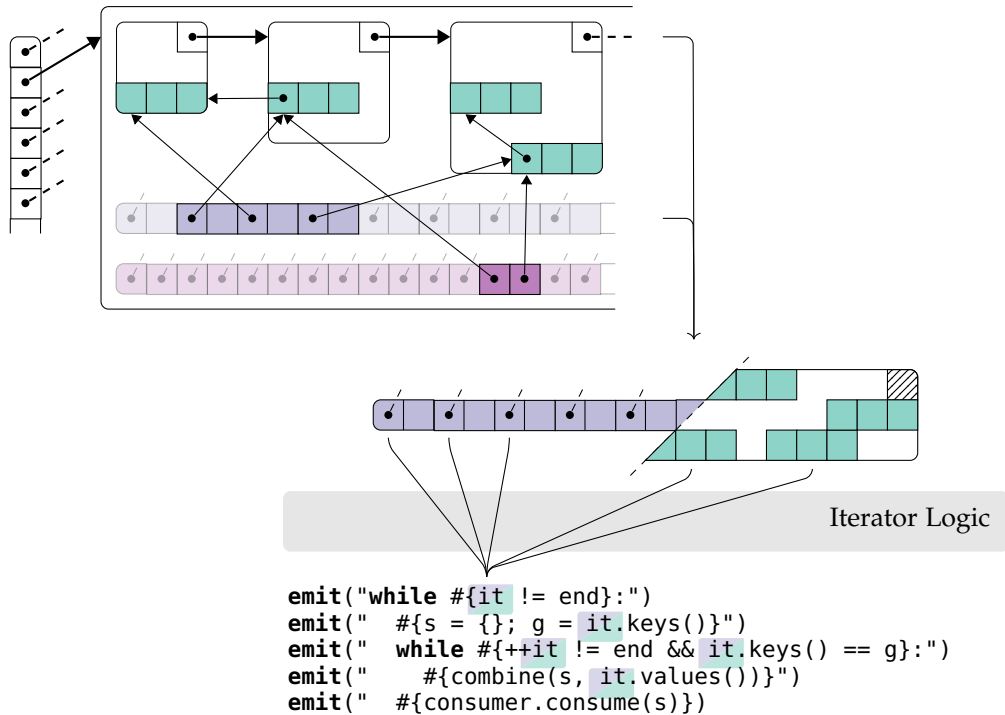


Figure 17: Tuple buffer and translator code that accesses sorted key ranges through iterator abstraction at query compile time.

However, in-place sorting also becomes inefficient with an increasing tuple size since the overhead of copying wide tuples overshadows the better cache efficiency. A common alternative is to sort a vector of pointers (or tuple identifiers) instead. These indirection vectors suffer from scattered memory accesses, but feature a constant entry size that will be beneficial once tuples get larger. This contrast leads to tradeoff between cache efficiency and robustness which oftentimes favors the latter. We instead combine the best of both worlds by introducing a third option that we call the *permutation vector*. A *permutation vector* is a sequence of entries that consist of the original tuple address followed by copied key attributes. This preserves the high efficiency of key comparisons in the operators SORT, ORDAGG, and WINDOW at the cost of a slightly more expensive vector construction.

Figure 17 shows a **chunk list**, a **permutation vector**, and a **hash table** of a single tuple buffer partition. The right-hand side of the figure lists an exemplary translation code for the ORDAGG operator. The code generates a loop over a sequence of tuples that aggregates key ranges and passes the results to a consumer. Keys and values are loaded through an iterator logic that abstracts the buffer access at query compile time. This way, the operator does not need

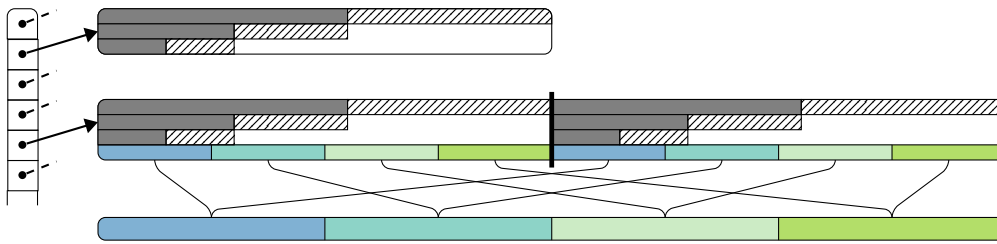


Figure 18: Morsel-Driven Quicksort and merge of two partitions. One partition is split eagerly to increase parallelism.

to be aware of either chunks or permutation vectors but can instead rely on the iterator to emit the appropriate access code.

4.4.3 Sorting

Sorting often dominates the evaluation of order-sensitive aggregate functions. Hence, the `SORT` operator is a central pillar for the fast evaluation of advanced aggregates. Our implementation is based on the *BlockQuicksort* algorithm by Edelkamp and Weiß [33]. *BlockQuicksort* extends a modified Introsort [86, 108] with a partitioning function that avoids branch mispredictions through block wise pivot comparisons. We adapt it for compiled query plans by using generated block comparison and block swap functions.

BlockQuicksort offers a good single-threaded sort performance but does not scale well to multiple cores. It is tempting to place this burden on the buffer partitioning by running the single-threaded sort algorithm on each buffer partition independently. This, however, is very vulnerable to data skew that cannot be mitigated easily in the presence of arbitrarily-sized groups. Even some moderate amount of data skew may stall the parallel query execution until the largest (straggler) partition is processed. We instead propose to embody the principles of *Morsel-Driven Parallelism* [71] into *BlockQuicksort*.

Morsel-Driven Parallelism elastically schedules small fragments of work on a dynamic number of worker threads. This eliminates any upfront scheduling decisions and allows the database system to adapt to data skew, varying system workloads, and NUMA topologies. In case of *BlockQuicksort*, however, we cannot just schedule arbitrary input fragments since the partitioning phases cannot be subdivided without introducing later merges. This leads to a tradeoff between fragment granularity and bookkeeping complexity.

We instead follow the observation that *BlockQuicksort* itself *generates* independent fragments quickly. *BlockQuicksort* uses repeated partitioning steps,

similar to textbook Quicksort implementations, to recursively sort ranges left and right of chosen pivot elements. We propose to adapt these partitioning steps by recursively sorting left ranges as usual but pushes the right ranges on an explicit stack that is shared with other threads. This will provide $O(\log n)$ *fragments* of each participating thread until every buffer partition is sorted. A Morsel-Driven scheduler can then *steal* those ranges and assign them to idle worker threads.

This adjustment makes BlockQuicksort elastic at very low cost. However, data skew can still limit the speedup because threads may have to wait for the very first partitioning steps to finish. We mitigate this effect by splitting overly large partitions upfront. This will increase the number of initial fragments at the cost of a following k-way merge. Morsel-Driven BlockQuicksort will therefore sort independently at very little overhead if buffer partitions are numerous and of roughly equal size. In extreme cases, the algorithm will perform an additional k-way merge afterwards. These two adjustments allow the algorithm to scale well with both the number of cores and the number of buffer partitions.

Figure 18 illustrates the Morsel-Driven BlockQuicksort using two buffer partitions. Each partition shows the ranges that are either sorted uninterruptedly, or shared with the other threads. The partition at the bottom is significantly larger than the partition on top and is therefore split in half upfront. This allows processing the very first morsels with three threads instead of two. Once all ranges are sorted, we select three global separators within the split partition and perform a parallel two-way merge afterwards.

4.4.4 Aggregation

The framework uses the three aggregation operators ORDAGG, WINDOW, and HASHAGG. In Figure 17, we already outlined the ORDAGG operator, which generates compact loops over sorted tuples and computes aggregates without materializing any aggregation state. We use the ORDAGG operator whenever our input is already sorted since it spares us explicit hash tables. We further use ORDAGG to efficiently evaluate nested aggregates such as $\text{SUM}(x - \text{MEDIAN}(x))$. Since all values are materialized, ORDAGG can compute the nested aggregates by scanning the key range repeatedly. Traditional operators are here forced to write back the result of the median into every single row and then compute the outer aggregate using a hash join. The LOLEPOPs therefore not only spare us

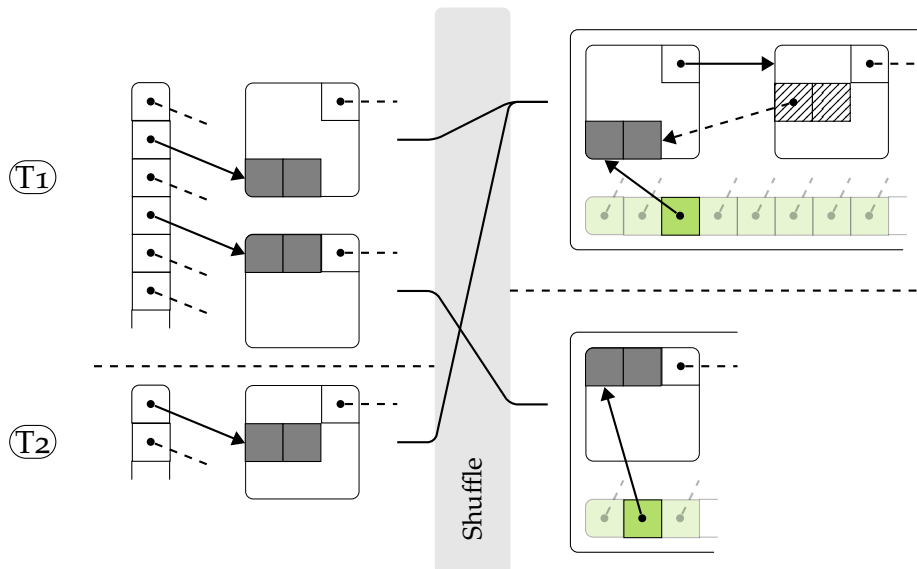


Figure 19: Two-phase hash aggregation with two threads. The hash tables on the left are fixed in size while the hash tables in green grow dynamically.

the hash tables but also the additional result field which will positively affect the sort performance.

The second aggregation operator is `WINDOW`. Algorithmically, its implementation closely follows the window operator described by Leis et al. [73]: Within our DAG, however, the materialization, partitioning and sorting of values is delegated to other LOLEPOPs and is no longer the responsibility of the `WINDOW` operator. Instead, the operator begins with computing the segment trees for every hash partition in parallel. Afterwards, it evaluates the window aggregates for every row and reuses the results between window frames wherever possible. We additionally follow the simple observation that segment trees can be computed for many associative aggregates at once, independent of their frames, as long as they share the same ordering. A single `WINDOW` operator therefore computes multiple frames in sequence to share the segment aggregation and increase the cache-efficiency.

The third aggregation operator, `HASHAGG`, is illustrated in Figure 19. `HASHAGG` adopts a two-phase hash aggregation [101, 71]. We first consume tuples of an incoming pipeline and compute partial aggregates in fixed-size thread-local hash tables. These first hash tables use chaining, and we allocate its entries in a thread-local partitioned tuple buffer. However, we do not maintain the hash table entries in linked lists, but instead simply replace the previous entry whenever the group keys differ. This will effectively produce a sequence of partially aggregated non-unique groups that have to be merged afterwards.

The efficiency of this operator roots in the ability to pre-aggregate most of its input in these local hash tables that fully reside in the CPU caches. That means that if the overall number of groups is small or if group keys are not too spread out across the relation, most of the aggregate computations will happen within this first step, which scales very well. Afterwards, the threads merge their hash partitions into a large buffer by simply concatenating the allocated chunk lists. These hash partitions are then assigned to individual threads and merged using dynamically-growing hash tables.

4.4.5 Partitioning

The `PARTITION` operator consumes an unordered tuple stream and produces a tuple buffer with a configurable number of, for example, 1024 hash partitions. Every single input tuple is hashed and allocated within a thread-local tuple buffer first. Once the input is exhausted, the thread-local buffers are merged across threads similar to the merging of hash partitions in the `HASHAGG` operator. Afterwards, the partition operator checks whether any of the following LOLEPOPs has requested to modify the buffer in-place. If that is the case, the partition operator compacts the chunk lists into a single chunk per partition. We deliberately introduced this additional compaction step to simplify the buffer modifications. The alternative would have been to implement all in-place modifications in a way that is aware of chunk-lists. This is particularly tedious for algorithms like sorting and also makes the generated iterator arithmetic in operators like `WINDOW` and `ORDAGG` more expensive.

4.4.6 Combine

The `COMBINE` operator joins unique groups on their group keys. Consider, for example, a query that pairs a distinct with a non-distinct aggregate. Both aggregates are computed in different `HASHAGG` LOLEPOPs but need to be served as single result group. The `COMBINE` operator joins an arbitrary number of input streams with the assumption that these groups are unique. We simply check for every incoming tuple whether a group with the given key already exists. If it is, we can just update the group with the new values and proceed. Otherwise, we materialize the group into a thread-local tuple buffer. After every pipeline, the `COMBINE` operator merges these local buffers and rehashes the partitions if necessary.

4.5 EVALUATION

In this section, we experimentally evaluate the planning and execution of advanced SQL analytics in Umbra using LOLEPOPs. We first compare the execution times of advanced analytical queries with Hyper, a database system that implements aggregation using traditional relational algebra operators. We then analyze the impact of aggregates on five TPC-H queries with a varying number of joins. Additionally, we show the performance characteristics of certain LOLEPOPs based on four execution traces. Our experiments have been performed on an Intel Core i9-7900X with 10 cores, 128 GB of main memory, LLVM 9, and Linux 5.3.

4.5.1 Comparison with other Systems

We designed a set of queries to demonstrate the advantages of our framework over monolithic aggregation operators. We chose the main-memory database system HyPer as reference implementation for *traditional* aggregation operators since it also employs code generation with the LLVM framework for best-of-breed performance in analytical workloads. The design of Umbra shares many similarities with HyPer besides the query engine which allows for a fair comparison of the execution times. Both systems rely on Morsel-Driven Parallelization [71] and compile queries using LLVM [87].

The database systems PostgreSQL and MonetDB were excluded due to their lacking performance for basic aggregates. The following table compares the execution times between HyPer, PostgreSQL and MonetDB for an associative aggregate, an ordered-set aggregate and a window function as well as for grouping sets with two group keys. Our queries represent complex and composed versions of these aggregates and will increase the margin between the systems further.

The experiment differs from benchmarks such as TPC-H or TPC-DS in that the queries are not directly modeling a real-world interaction with the database. We instead define queries that only aggregate a single base table without further join processing. We focus on the relation *lineitem* of the benchmark TPC-H since it is well-understood and may serve as placeholder for whatever analytical query precedes. This does not curtail our evaluation since those operators usually form the very top of query plans and any selective join would only

Table 2: Execution times in seconds of queries with simple aggregates in HyPer, PostgreSQL and MonetDB.

Query	HyPer	PgSQL	MonetDB
SUM(q) GROUP BY k	0.50	4.03	0.64
SUM(q) GROUP BY ((k,n),(k))	0.55	42.31	4.77
PCTL(q,0.5) GROUP BY k	0.89	32.96	10.19
ROW_NUMBER() PARTITION BY k ORDER BY q	0.87	26.58	10.36

n=linenumber q=quantity k=supkey

reduce the pressure on the aggregation logic. Our performance evaluation comprises 18 queries across five different categories. Table 3 shows the execution times using Umbra and HyPer with 1 and 20 threads and the factors between them.

Queries 1, 2, and 3 provide descriptive statistics for the single attribute `extendedprice` with a varying number of aggregation functions. The aggregates in all three queries have to be optimized as a whole since they either share computations or favor different evaluation strategies. Query 2 presents a particular challenge for monolithic aggregation operators since the function `percentile` (PCTL) is not associative. The associative aggregates `SUM`, `COUNT`, and `VAR_SAMP` can be computed on unordered streams and can be aggregated eagerly in thread-local hash tables. Non-associative aggregates like PCTL, on the other hand, require materialized input that is sorted by at least the group key. HyPer delegates this computation to the Window operator and computes the associative aggregates using a subsequent hash-based grouping. Umbra computes all aggregates on the sorted key range using the `ORDAGG` operator, which spares us the hash tables.

Queries 4, 5, 6, and 7 target the scalability of ordered-set aggregates. All four queries are dominated by the time it takes to sort the materialized values and therefore punish any unnecessary reorderings. The databases have to optimize the plan with respect to the ordering constraints to eliminate redundant work in query 5 and 6. Query 7 additionally groups by the attribute `linenumber` which contains only seven distinct values across the relation. HyPer does not sort partitions in parallel and is therefore considerably slower when scaling to 20 threads.

Queries 8, 9, 10, 11, and 12 analyze grouping sets that introduce a significant complexity in the aggregation logic by combining different group keys. This offers potential performance gains through reaggregation of associative aggregates and stresses the importance of optimized sort orders. HyPer only sup-

Table 3: Execution times in seconds for advanced SQL queries on the TPC-H lineitem table (scale factor 10).

#	Aggregates	1 thread		20 threads				
		Umbr	HyPer	Umbr	HyPer			
1	SUM(e), COUNT(e), VAR_SAMP(e)	GROUP BY k	3.10	4.73	1.53	0.37	0.60	1.62
2	↳, PCTL(e,0.5)	GROUP BY k	4.32	9.36	2.17	0.47	0.96	2.03
3	COUNT(e), COUNT(DISTINCT e)	GROUP BY k	9.61	127.63	13.28	1.21	26.52	21.90
4	PCTL(e,0.5)	GROUP BY k	4.00	8.88	2.22	0.43	0.92	2.14
5	↳, PCTL(e,0.99)	GROUP BY k	4.02	12.66	3.15	0.42	1.40	3.31
6	↳, PCTL(q,0.5), PCTL(q,0.9)	GROUP BY k	6.48	22.39	3.46	0.64	2.68	4.20
7	PCTL(e,0.5), PCTL(q,0.5)	GROUP BY n	6.74	21.93	3.25	0.93	19.85	21.36
8	SUM(q)	GROUP BY ((k,n),(k),(n))	2.30	10.73	4.66	0.28	1.09	3.96
9	SUM(q)	GROUP BY ((k,s,n),(k,s),(k,n),(n))	2.63	16.37	6.22	0.42	1.71	4.09
10	PCTL(q,0.5)	GROUP BY ((k,n),(k))	2.43	18.11	7.46	0.24	1.85	7.56
11	PCTL(q,0.5)	GROUP BY ((k,s,n),(k,s),(k))	2.77	27.78	10.05	0.31	2.89	9.44
12	PCTL(q,0.5)	GROUP BY ((k,n),(k),(n))	1.97	26.60	13.50	0.52	10.43	20.20
13	LEAD(q), LAG(q)	PARTITION BY k ORDER BY r	8.33	13.69	1.64	0.97	1.46	1.50
14	↳, CUMSUM(q)	PARTITION BY k ORDER BY d	12.77	19.05	1.49	1.56	2.27	1.46
15	CUMSUM(q)	PARTITION BY n ORDER BY d	5.10	12.32	2.42	0.89	10.93	12.29
16	PCTL(e - PCTL(e,0.5),0.5)	GROUP BY k	6.35	12.39	1.95	0.69	1.44	2.07
17	PCTL(SUM(q), 0.5)	GROUP BY k	1.58	4.08	2.58	0.20	0.52	2.62
18	SUM(POW(LEAD(q) - q,2)) / COUNT(*)	GROUP BY k	5.63	10.90	1.94	0.58	1.09	1.89

e=extendedprice n=linenumber s=linestatus o=orderkey p=partkey
 q=quantity r=receiptdate k=suppkey d=shipdate m=shipmode

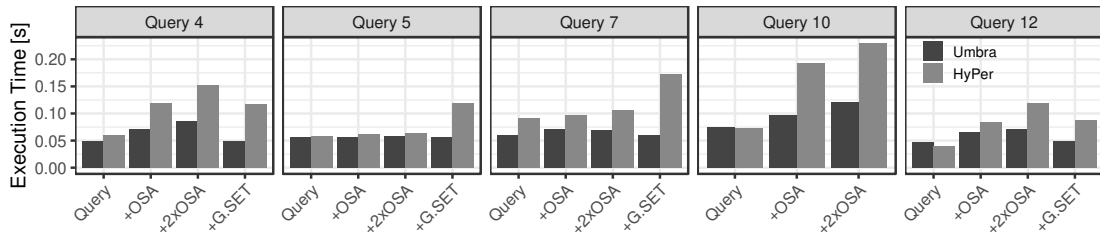


Figure 20: Execution times of five TPC-H queries at scale factor 10 with and without additional aggregates.

ports grouping sets by computing the different groups independently and combining the results using `UNION ALL`. With LOLEPOPs, we instead start grouping by the longest group keys first and then reaggregate key prefixes whenever necessary. In query 8, for example, we first group by `(suppkey, linenumber)` and then reaggregate the results by `suppkey` afterwards. Queries 10, 11, and 12 use the percentile function and emphasize the sort optimizations of our framework. We compute the queries 10 and 11 efficiently on a single buffer that is partitioned by attribute `suppkey`. We reorder the buffer by the constraints arranged in decreasing lengths, i.e., `(suppkey, linenumber, quantity)` followed by `(suppkey, quantity)` for query 10. Query 12 adds `(linenumber)` as additional group key which will again penalize systems that sort key ranges in a single-threaded fashion.

Queries 13, 14, and 15 target the scaling (in terms of number of expressions) of window queries. Query 13 combines the two window functions `LEAD` and `LAG` that can be evaluated on the same key ranges. Query 14 adds a cumulative sum on a different ordering attribute which favors an efficient reordering of the previous key range. Query 15 partitions by `linenumber` again to underline the importance of parallel sorting for all flavors of ordered aggregation.

Queries 16, 17, and 18 compose complex aggregates from window and grouping aggregates. Query 16 computes the Median Absolute Deviation (MAD) function that we described as advanced aggregate in Section 4.3.3. It first computes a median m of the attribute `extendedprice` as window aggregate and then reorders the buffer to compute the median of `(extendedprice - m)` as ordered-set aggregate. With LOLEPOPs, we can explicitly reorder the partitioned buffer by the first computed median aggregate and then compute the second median with a `ORDAGG` operator. Query 18 computes the also aforementioned function Mean Square Successive Difference (MSSD) that sums up the square difference between the window function `LEAD` and a value. This time, we do not need to reorder values but

can directly compute the result on the sorted key range using ORDAGG. They query also shows that the performance of *traditional* aggregation operators is sometimes saved by coincidence due to almost-sorted tuple streams. In HyPer, the WINDOW operator streams the key ranges to the hashing GROUP BY almost in order, improving the effectiveness of thread-local pre-aggregation.

In summary, these 18 queries show scenarios that occur in real-world workloads and already profit from the optimizations on a DAG of LOLEPOPs. These optimizations are quite difficult to implement in relational algebra, but can be broken up into composable blocks with LOLEPOPs.

4.5.2 Advanced Aggregates in TPC-H

We next analyze the performance impact of advanced aggregates on TPC-H. Figure 20 shows the execution times of the TPC-H queries 4, 5, 7, 10, and 12 with and without additional aggregates at scale factor 10. The modifications of the individual queries only consist of up to two additional ordered set aggregates with different orderings or a prefix of the group key as additional grouping set.

Query 5 and 7 contain five joins that pass only few tuples to the topmost GROUP BY operator. Both queries are dominated by join processing and the additional ordered-set aggregates, percentiles of `l_quantity` and `l_discount`, have an insignificant effect on the overall execution times. Yet, the additional grouping by either `n_name` or `l_year` doubles the execution times in HyPer since the joins are duplicated using UNION ALL. This suggests that, even without Low-Level-Plan operators, a system should at least introduce temporary tables to share the materialized join output between different GROUP BY operators.

Query 4 only contains a single semi join that filters 500,000 tuples of the relation orders. This increases the pressure on the aggregation logic which results in a slightly faster execution with Low-Level-Plan operators. These differences are further pronounced in the modified queries computing additional percentiles of `o_totalprice` and `o_shippriority` and grouping by `o_orderstatus`. Query 12 behaves very similar to query 4 and aggregates 300,000 tuples. HyPer is slightly faster when computing the original aggregates but loses when adding the percentiles `l_quantity` and `l_discount` or grouping by `l_linestatus`.

Query 10 aggregates over one million tuples produced by three joins and is also slightly faster in HyPer. However, Umbra outperforms HyPer by a factor of almost two when additionally computing the percentiles `l_quantity` and `l_discount`. This is attributable to the high number of large groups that are accumulated by the aggregation operator. The aggregation yields over 300,000 groups that are reduced with a following top-k filter. As a result, traditional hash-based aggregation suffers from the cache-inefficiency of larger hash tables. Umbra loses slightly against HyPer when computing the single sum but wins as soon as the ordered-set aggregates can eliminate the hash aggregation entirely.

The experiment demonstrates that minor additions to the well-known TPC-H queries such as adding a single ordered-set aggregate or appending a grouping set suffice to unveil the inefficiencies of monolithic aggregation operators. It also shows that queries may very well be dominated by joins, leaving only insignificant work for a final summarizing aggregation operator. In such cases, the efficiency of the aggregation is almost irrelevant which is not changed by introducing LOLEPOPs.

4.5.3 LOLEPOPs in Action

In a next step, we illustrate the different performance characteristics of LOLEPOPs based on two different example queries. Both queries target the relation `lineitem` of TPC-H. We execute the queries at scale factor 0.5 with four threads and reduce the number of partitions in tuple buffers to 16 to make morsels graphically distinguishable. Figure 21 shows precise timing information about the morsels being processed.

Query 1 computes the aggregate `SUM` grouped by the grouping sets `(suppkey, linenumber)`, `(suppkey)`, and `(linenumber)`. It is significantly faster than the second query although it groups the input using three different `HASHAGG` operators. Umbra computes these grouping sets efficiently by pre-aggregating the 3 million tuples of the **first** scan pipeline by `(suppkey, linenumber)`. The **second** pipeline then merges these partial aggregates into 35,000 groups using dynamic hash tables for each of the 16 partitions. The **third** pipeline scans the results afterwards, re-aggregates them by `suppkey` and `linenumber`, and passes them to the `COMBINE` operator. All remaining pipelines are barely visible since they operate on only a few tuples. The plan of this query corresponds to query ① in ??.

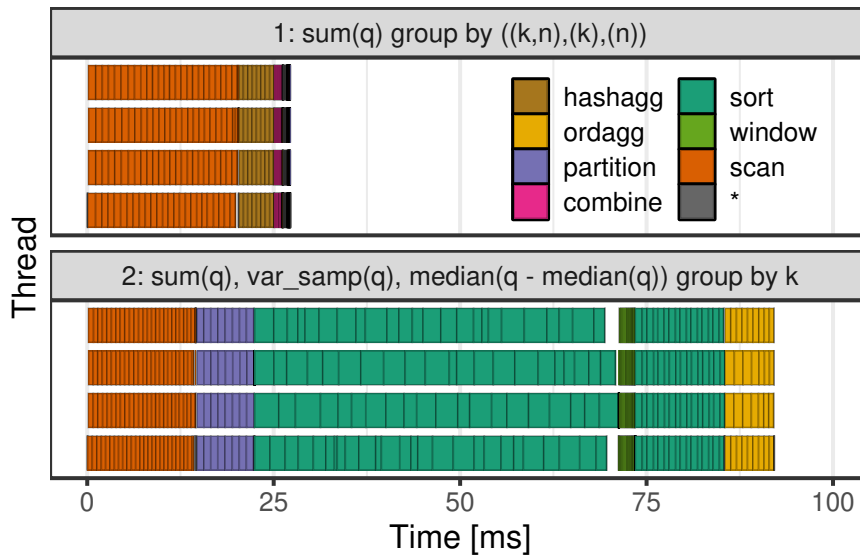


Figure 21: Execution traces of two queries on the TPC-H schema at scale factor 0.5 with four threads and 16 buffer partitions.

Query 2 computes the associative aggregates SUM and VAR_SAMP, as well as the Median Absolute Deviation that was introduced as advanced aggregate in Section 4.3. We include it as execution trace in this experiment to illustrate the advantages of sharing materialized state between operators. Umbra evaluates this query by computing the nested median as window expression and then reordering the results in place. In contrast to the previous query, the **first** pipeline is now faster since it only materializes the tuples into 16 hash partitions. Thereafter, the **compaction** merges the chunk list of each hash partition into single chunks that enable the in-place modifications. The **fourth** pipeline represents the window operator that computes the median for every key range and stores the result in every row. The **following** pipeline then reorders the partitioned buffer by this median value. It is significantly faster than the first sort since the hash partitions are already sorted by the key. The **last** pipeline then iterates over the sorted key-ranges and computes the three aggregates at once.

4.5.4 Adaptive Sorting

We have seen that aggregation is oftentimes dominated by the sorting of materialized values. As a result, we introduced a flexible iterator abstraction and the permutation vector in Section 4.4 to make sorting more cache efficient.

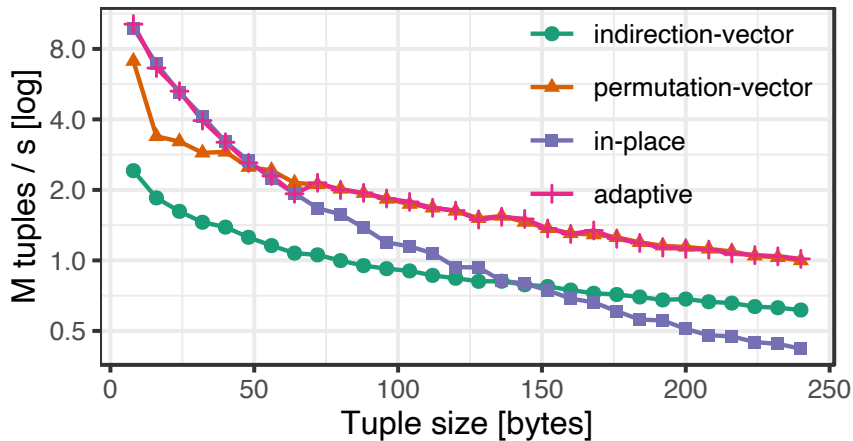


Figure 22: Sort performance for varying tuple sizes using different access methods.

In-place sorting can be significantly faster than sorting a vector of pointers since the overall sort performance is driven by the cache efficiency. Vectors of addresses are a very conservative approach to sorting that is robust against growing tuple sizes. If tuples get overly large, the cache efficiency of in-place sorting will evaporate and the excessive copies will add detrimental effects. In this experiment, we analyze the impact of a growing tuple size on the sort performance.

We generate a relation that contains 100 million tuples with 40 attributes of random 4-byte integers. We sort the relation by a single attribute with the four different access modes, in-place, a permutation vector, a vector of pointers, and an adaptive approach that picks the access mode based on the tuple size. We select a varying number of values and scan the sorted buffer once after sorting to incorporate the effects of indirections on the buffer accesses. Figure 22 shows the results of the experiment. In-place sorting turns out to be the fastest choice for all tuple sizes up to 64 bytes. The permutation vector is around 60% faster than the indirection vector across all tuple sizes. The raw indirection vector is the worst option in this experiment although this is partly attributable to the small key size. Growing key sizes favor the indirection vector over the permutation vector as soon as the cache efficiency of the key access gets overruled by the costs for copying. All three options therefore have their own sweet spots where they become the most efficient solution. The experiment further suggests that an adaptive mode does not require excessive fine-tuning to take advantage of cache-efficient sorting for small tuple sizes.

4.6 RELATED WORK

Research on optimizing in-memory query processing on modern hardware has been growing in the past decade. However, in comparison to joins [14, 4, 5, 6, 69, 9, 7, 106, 8, 38], work on advanced statistics operators is relatively sparse. Efficient aggregation is described by a number of papers [101, 97, 71, 35, 85]. However, these papers omit to discuss how to implement DISTINCT aggregates, which are significantly more complicated than simple aggregates, in particular, when implemented using hashing. There is even less work on window functions: The papers of Leis et al. [73] and Wesley et al. [126] are the only one that describe the implementation of window functions in detail. Cao et al. [19] present query optimization techniques for queries with multiple window functions (e.g., reusing existing partitioning and ordering properties), which are also applicable and indeed are directly enabled by our approach. Except for Xu et al. [135], much work on optimizing sort algorithms for modern hardware [24, 33] has focused on small tuple sizes. Grouping sets have been proposed by Gray et al. [43] in 1997, and consequently there have been many proposals for optimizing the grouping order: Phan and Michiardi's [94] fairly recent paper offers a good overview. We are not aware of any research papers describing how to efficiently implement ordered-set aggregates in database systems. Even more importantly, all these papers focus on a small subset of the available SQL functionality and do not discuss how to efficiently execute queries that contain multiple statistical expressions like the introductory example in Section 4.1. Given the importance of statistics for data analytics, this chapter fills this gap by presenting a unified framework that relies on many of the implementation techniques found in the literature.

Our approach uses the notion of LOw-LEvel Plan Operators (LOLEPOP), which was proposed in 1988 by Lohman [80] and is itself based on work by Freytag [39]. According to them, a LOLEPOP may either be a traditional relational operator (e.g., join or union) or a low-level building block (e.g., sort or ship). The result of a LOLEPOP may either be a buffer or a stream of tuples. Furthermore, a LOLEPOP may have *physical properties* such as an ordering. A concept similar to LOLEPOPs was very recently described by Dittrich and Nix [32], who focus on low-level query optimizations. They introduce the concept of Materialized Algorithmic Views (MAVs), that represent materialized results at various granularity levels in the query plan. We understand our LOLEPOPs to be an instantiation of MAVs at a granularity that is slightly lower than relational algebra. Our framework further represents aggregates as

directed acyclic graphs and therefore might benefit from existing research on parallel dataflows [34]. There are also some similarities with low-level algebras [95, 89] and compilation frameworks [28, 27, 17]. However, these papers generally focus on simple select-project-join queries and on portability across heterogeneous hardware, but do not discuss how to translate and execute complex statistical queries.

The optimization of query plans with respect to *interesting sort orders* dates back to a pioneering work of Selinger et al. [109]. They consider a sort order *interesting* if it occurs in GROUP BY or ORDER BY clauses or if it is favored by join columns. These sort orders are then included during access path selection, for example, to introduce *merge joins* for orders that are required anyway. AWS Redshift is a distributed commercial system that uses *interesting* orders to break up certain aggregations at the level of relational algebra operators. Redshift introduces the operator *GroupAggregate* that consumes materialized tuples from a preceding *Sort* operator to compute ordered-set aggregates more efficiently. Our system generalizes this idea and performs access path selection with *interesting* orderings and partitionings to derive LOLEPOPs for all kinds of complex aggregation functions.

4.7 SUMMARY

The SQL standard offers a wide variety of statistical functionalities, including associative aggregates, distinct aggregates, ordered-set aggregates, grouping sets, and window functions. In this chapter we argue that relational algebra operators are not well suited for expressing complex SQL queries with multiple statistical expressions. Decomposing complex expressions into independent relational operators may lead to sub-optimal performance because query execution is generally derived directly from this execution plan. We instead propose a set of low-level plan operators (LOLEPOPs) for SQL-style statistical expressions. LOLEPOPs can access and transform buffered intermediate results and, thus, allow reusing computations and physical data structures in a principled fashion. Our framework subsumes the sort-based and hash-based aggregation as well as several non-statistical SQL constructs like ORDER BY and WITH. We presented our LOLEPOP implementations and integrated our approach into the high-performance database system Umbra. The experimental comparison against the HyPer shows that LOLEPOPs improve the performance of queries with complex aggregates substantially.

5

REDUCING LATENCY IN COMPILING QUERY ENGINES

Excerpts of this chapter have been published in [64, 66].

Initial experiments were performed in [62].

5.1 INTRODUCTION

Chapter 2 describes compiling query engines as popular approach for executing analytical queries. The main advantage of translating SQL to machine code is, of course, efficiency. By generating code for a given query, compilation avoids the interpretation overhead of traditional execution engines and thereby achieves much higher performance.

One obvious drawback of generating machine code is compilation time. Consider, for example, the following meta data query:

```
SELECT c.oid, c.relname, n.nspname
FROM pg_inherits i
JOIN pg_class c ON c.oid = i.inhparent
JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE i.inhrelid = 16490
ORDER BY inhseqno
```

This query touches only a very small number of tuples, which means that its execution time is negligible (less than 1 millisecond in HyPer). However, before HyPer can execute this query, it needs to compile it to machine code. With optimizations enabled, LLVM takes 54ms to compile this query. In other words, compilation takes 50 times longer than execution. Assuming a workload where similar queries are executed frequently, 98% of the time will be wasted on compilation. And this query is still fairly small; compilation times can be much higher for larger queries. Compilation of the largest TPC-DS query, for example, takes close to 1 second. Of course, for large data sizes, compilation does pay off as the resulting code is much more efficient than interpretation.

In this work, we focus on database systems that compile queries to LLVM IR (“Intermediate Representation”), which is afterwards compiled to machine

code by the LLVM compiler backend. This approach offers the same machine code quality as compiling to C/C++, while reducing compilation time by an order of magnitude [87]. The compilation times of the LLVM compiler may be low enough for some workloads, for example those consisting of long-running ad hoc queries or pre-compiled stored procedures. For other applications, however, long compilation times are a major problem.

The example query shown above is one of the queries sent by the PostgreSQL administration tool *pgAdmin*. We identified in [62], that *pgAdmin* executes dozens of complex queries (up to 22 joins), all of which access only very small meta data tables. Compiling these queries causes perceptible and unnecessary delays. Caching the machine code (e.g., after stripping out constants) might improve subsequent executions, but would not improve the initial user experience. More generally, because the human perception threshold is less than a second, the additional latency caused by compilation can lead to a worse user experience for interactive applications. Finally, business intelligence tools occasionally generate extremely large queries (e.g., 1 MB of SQL text) [125], which de facto cannot be compiled with standard compilers.

For the workloads just mentioned, the user experience of a compilation-based engine can be *worse* than that of a traditional, interpretation-based engine (e.g., Volcano-style execution). Thus, depending on the query, one would sometimes prefer to have a compilation-based engine and sometimes an interpretation-based engine. Implementing two query engines in the same system, however, would involve disproportionate efforts and may cause subtle bugs due to minor semantic differences. In this work, we instead propose an adaptive execution framework that is principally based on a single compilation-based query engine, yet integrates interpretation techniques that reduce the query latency. The key components of our design are a (i) fast bytecode interpreter specialized for database queries, (ii) a method of accurately tracking query progress, and (iii) a way to dynamically switch between interpretation and compilation. Without relying on the notoriously inaccurate cost estimates of query optimizers, this dynamic approach enables the best of both worlds: Low latency for small queries and high throughput for long-running queries.

This thesis extends the adaptive execution framework proposed in [62]. It is directly applicable to many compilation-based systems and is non-invasive, i.e., the query engine itself does not have to be rewritten. The system always generates LLVM IR code for the incoming query but does not immediately compile it to machine code. Rather, it adaptively ensures that the query is

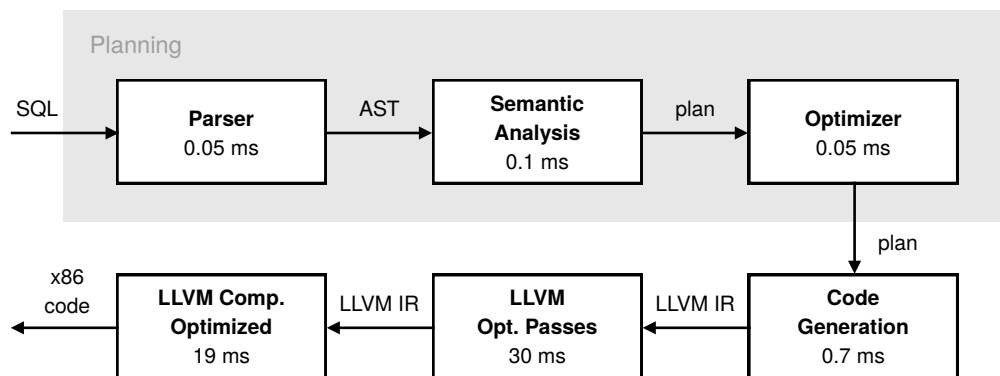


Figure 23: Architecture of compilation-based query engines.

executed as fast as possible using runtime feedback and without relying on cost estimates from the query optimizer.

This chapter is organized as follows: Section 5.2 discusses how compilation-based query engines compile queries to machine code and the challenges these systems face. Section 5.3 then describes our adaptive execution framework, which dynamically switches from interpretation to compilation. One crucial component of that design is a bytecode interpreter, which we introduce in Section 5.4. In Section 5.5, we compare the performance of the interpreter with machine code, and evaluate our adaptive execution framework. After discussing related work and alternative approaches for reducing compilation time in Section 5.6, we summarize the chapter in Section 5.7.

5.2 QUERY EXECUTION VIA COMPILATION

Executing a SQL query in a relational database system involves a complex multi-step process that is illustrated in Figure 23. The SQL text is first parsed into an abstract syntax tree (“Parser”). The AST is translated into an unoptimized query plan (“Semantic Analysis”) that is optimized afterwards (“Optimizer”). Traditional engines directly execute this query plan (e.g., using Volcano-style iteration). Compilation-based engines, in contrast, translate the optimized query plan into some imperative, low-level, machine-independent language (“Code Generation”) that is optimized again (“LLVM Opt. Passes”) and finally compiled to machine code (“LLVM Comp. Optimized”). Some compilation-based systems have multiple intermediate languages between the relational algebra and the low-level imperative representation (LLVM IR in our case). This does not really affect our discussion, as machine code genera-

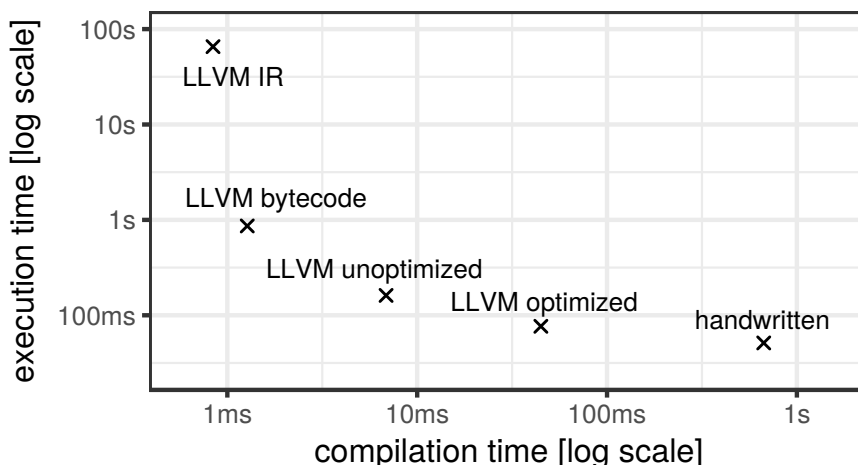


Figure 24: Single-threaded query compilation and execution time for different execution modes on TPC-H query 1 on scale factor 1.

tion generally takes longer than these additional phases. In the following, we describe the major challenges faced by compilation-based query engines that compile to machine code.

5.2.1 Latency vs. Throughput Tradeoff

This chapter is based on HyPer, which executes queries by compiling them to the LLVM IR. LLVM is a widely-used open source compiler framework for ahead-of-time and just-in-time compilation. Figure 23 shows the execution times of each stage for TPC-H query 1 in our system using LLVM. The numbers show that most time is spent in the final two LLVM compiler phases (“LLVM Opt. Passes” and “LLVM Comp. Optimized”), while the preceding code generation, query optimization, and analysis phases are negligible. Therefore, to optimize overall latency, we need to focus on making machine code generation cheaper (or avoid it completely).

Compilation time and execution time differ depending on the compiler and optimization settings used. Figure 24 shows the compilation and execution time of TPC-H query 1 on scale factor 1 under different settings¹. As the figure shows, LLVM has a similar throughput as the handwritten C++ query while requiring a much lower compilation time². Disabling all LLVM optimizations results in significantly lower compilation times at the cost of (slightly) higher

¹ The experimental setup is described in Section 5.5.

² Note that the handwritten version does not implement overflow checks, which explains its slightly faster runtime.

execution times. The figure also shows the built-in LLVM interpreter (“LLVM IR”), which directly interprets the LLVM IR module, and our bytecode-based interpreter (“LLVM bytecode”), which we describe in Section 5.4. These numbers show that only interpreters can achieve very low latency but, unsurprisingly, this is achieved by sacrificing throughput.

Figure 24 clearly shows that there is a tradeoff between latency and throughput. For long-running queries, compilation to machine code with a maximum optimization level is often preferable, while for quick queries an interpreter would be best. Unoptimized machine code lies in between and offers a good tradeoff between these two extremes. Depending on the complexity of the SQL query and the amount of data accessed, different execution schemes are optimal. In this work, we therefore propose to dynamically adapt the query execution by switching between a LLVM bytecode interpreter and the LLVM compiler with optional optimization passes.

Another relevant aspect is that not all code paths of a query are equally important. A query consisting of an in-memory hash join between a very small build relation and a large probe relation, might be best executed by interpreting the hash table build code and compiling the hash probe code. Thus, for different parts of the query, different execution modes can be ideal. Let us also note that compilers are single-threaded, while modern query engines are generally multi-threaded. Thus, while compilation is ongoing, all but one CPU cores are idle, whereas an interpreter could start utilizing all available cores much earlier.

5.2.2 Compiling Large Queries

A compilation time of 59ms for TPC-H query 1 may still be considered low enough for some applications. However, query 1 is still fairly small in terms of its resulting code size and larger queries take much longer to compile. The compilation time of the largest TPC-H and the largest TPC-DS query are 146ms and 911ms respectively.

Furthermore, we observed for very large, machine-generated queries (e.g., from business intelligence tools), that the compilation times grow super-linearly with the query size. The compilation times may thus become significantly longer and queries may not finish at all (see Section 5.5.5). While such queries are not common, any industrial-strength system must be able to

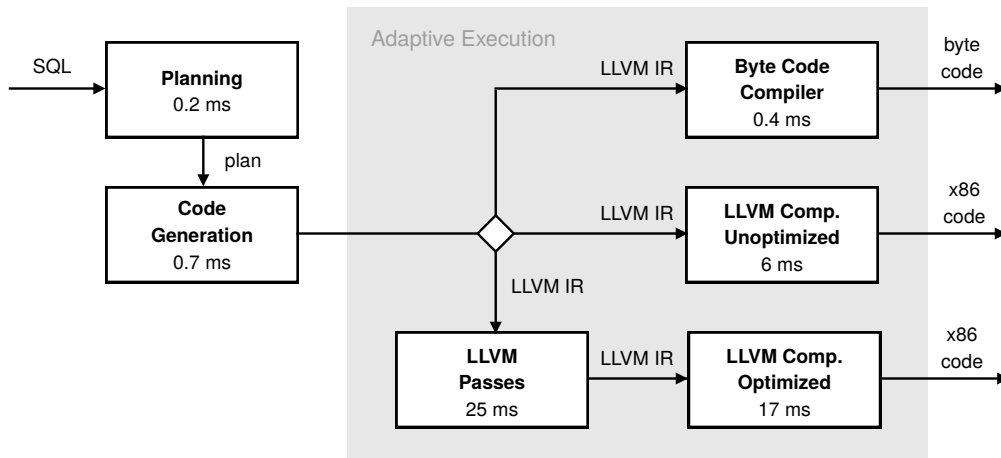


Figure 25: Execution modes and their compilation times.

execute them—in particular, since traditional database systems do not have this problem.

5.3 ADAPTIVE EXECUTION

We argue that compilation-based engines should support the 3 execution modes shown in Figure 25. Bytecode interpretation enables very low latency for quick queries, unoptimized machine code is a good tradeoff for medium-sized queries, and optimized machine code achieves peak throughput for long-running queries. A system that supports these modes, can provide an optimal user experience if it chooses the right mode for a given query.

One possible way to decide between the different execution modes is to rely on the cost estimates computed by the query optimizer. Cardinality estimates as well as cost models are often inaccurate [72, 74], which may result in unnecessary compilations or long-running queries being executed in the interpreter. The effects of a wrong decision can be severe as the bytecode interpreter can be slower by an order of magnitude and compilation can easily take hundreds of milliseconds. Furthermore, the compilation itself is single-threaded so compiling up-front leaves all remaining threads idle until the compilation is finished.

5.3.1 Overview

Our adaptive execution approach is dynamic, as we avoid any up-front decision about the execution mode. Instead, we always start executing every query

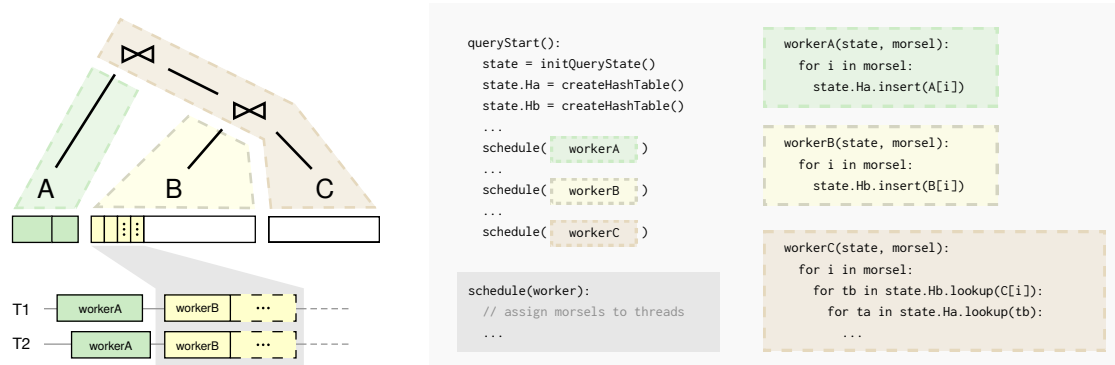


Figure 26: Illustration of query plan translation to pseudo code. *queryStart* is the main function. Each of the three query pipelines is translated into a *worker* function. The lower left corner shows that the work of each pipeline is split into small morsels that are dynamically scheduled onto threads.

using the bytecode interpreter and all available threads. We then monitor the execution progress to decide whether (unoptimized or optimized) compilation would be beneficial. If this is the case, we start compiling on a background thread, while the other threads continue the interpreted execution. Once compilation is finished, all threads quickly switch to the compiled machine code. Because all execution modes semantically execute the same instructions on the same data structures, no work is lost when switching between execution modes and the machine code can pick up where the interpreter left off.

Our approach is fine-grained. The tracking and the decision to compile is not done for the entire query, but for a specific query pipeline (e.g., an expensive hash table probe). Therefore, different pipelines might be executed using different execution modes. This can be better than any static up-front decision because optimized compilation is done only for very expensive parts of the query. As we show later, in a multi-threaded setting, it is also often beneficial to execute the same pipeline by consecutively running all 3 modes.

To implement our dynamic approach we need a number of mechanisms, which we describe in the following three sections. First, it must be possible to track the progress of a pipeline. Second, there must be a way to switch the execution from bytecode to compiled execution without losing any work. Finally, we need a model for deciding whether it is beneficial to switch to compilation.

5.3.2 Tracking Query Progress

Figure 26 illustrates the basic code structure for an example query plan and the code structure that our compiler generates. The entry point is the `queryStart` function, which, when called, executes the query. `queryStart` is mainly responsible for calling C++ initialization code and for launching the different pipelines of the query. It can be fairly large in size, but since this code is executed only once, it never pays off to compile it. The actual data-dependent parts of the query are always performed in the worker functions, each of which computes the result of one pipeline. The query plan in Figure 26 consists of 3 pipelines that are processed by the 3 worker functions `workerA`, `workerB`, and `workerC`.

This code structure has been chosen with multi-core parallelization in mind. Each worker function requires two arguments: the state (e.g., intermediate query processing hash tables) and a `morsel`, which determines the range of values to process. Intra-query parallelization is implemented by running multiple worker threads on the same worker function, but with different (non-overlapping) morsels (e.g., of a relation). In our parallelization framework, the threads use work stealing and the range for each invocation is fairly small (e.g., 10,000 tuples), which avoids thread imbalances.

This morsel-wise (or block-wise) execution has been identified as a fast model for intra-operator parallelism in main-memory databases [21, 71, 91] and is also just the right granularity at which the progress of a query can be monitored. After each morsel, worker threads consult a work-stealing data structure anyway. At this point, we added some extra monitoring code and timing information to keep track of how many morsels have been processed per pipeline. Additionally, we also record the total amount of work whenever a pipeline starts (e.g., the size of the relation or hash table being scanned). This information allows us to monitor the query progress.

5.3.3 Switching Between Execution Modes

With adaptive execution, a morsel represents the smallest unit of work that can be processed by the query engine. Besides being useful to track the progress of a query, these morsels also emerge as the crucial mechanism for dynamically switching between different execution modes. Processing a single morsel involves reading a specified range of values and operating on a shared state like a hash table. Providing both—range and shared state—as input pa-

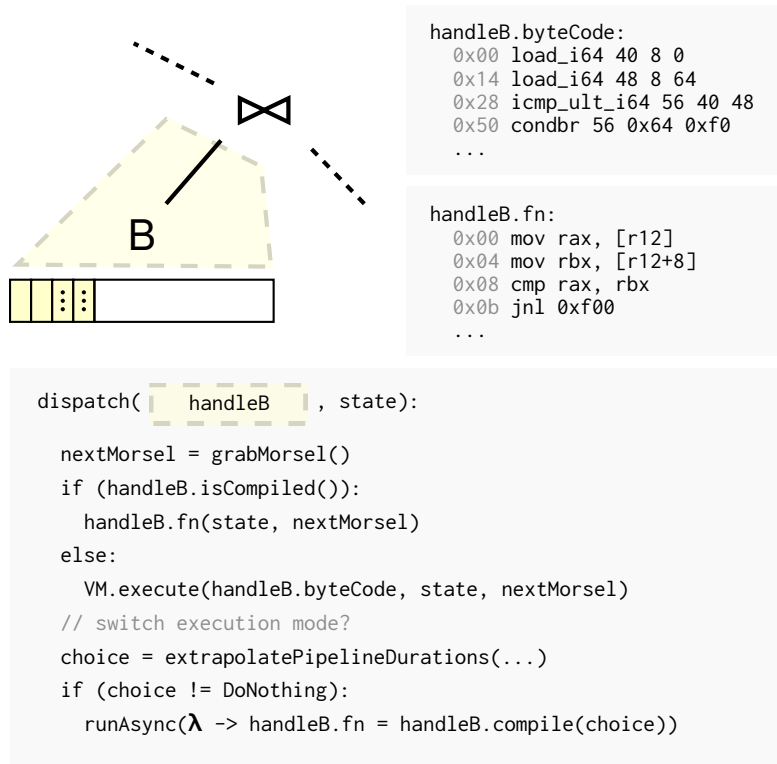


Figure 27: Switching on-the-fly from interpretation to execution. The dispatch code is run for every morsel.

rameters to a worker function simplifies the generated code and makes the processing of morsels independent from each other.

This independence enables us to choose the execution mode for an individual worker arbitrarily. It becomes semantically equivalent to process either all morsels, every second morsel, or no morsel with the bytecode interpreter and process the remainder with a compiled worker function. We can further compile a single worker function multiple times with different optimization levels to improve the throughput of the function step-by-step.

Figure 27 illustrates the integration of this concept into the morsel-driven parallelization framework. Instead of identifying a worker function by its memory address, we introduce an additional handle indirection. This object stores multiple variants of the same function. For every single morsel, we then choose the fastest available representation which could either be bytecode or an address to compiled machine code. Consequently, to change the execution mode, one only needs to set a function pointer in this handle object. Once set, all remaining morsels will be processed using the new variant enabling seamless transitions between execution modes.

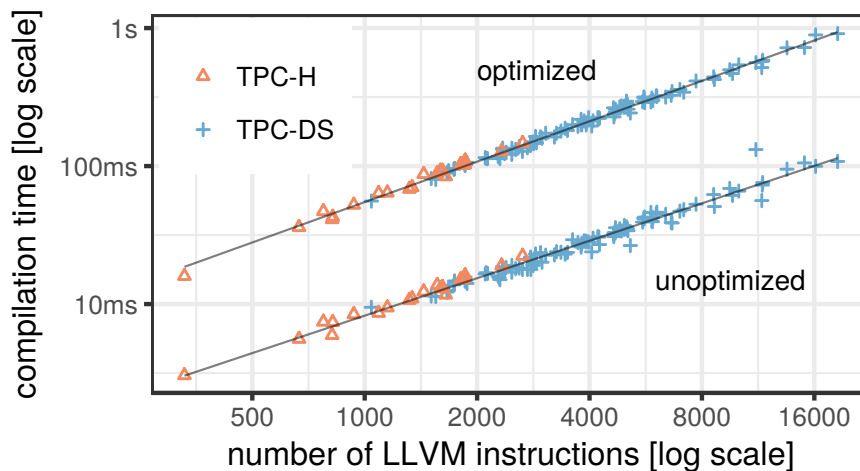


Figure 28: LLVM compilation time for (un-)optimized machine code for TPC-H and TPC-DS queries.

5.3.4 Choosing Execution Modes

We have already shown that a higher query throughput comes at the cost of a higher query latency. We therefore always start the execution of each worker function with the low-latency bytecode interpreter and compile it only if the need becomes evident. This, however, raises the question of how to determine when compilation is beneficial. To make this decision, we continuously evaluate the following options for every pipeline:

1. proceed with the current execution mode
2. compile the worker function to machine code without compiler optimizations (unoptimized)
3. compile the worker function to machine code with compiler optimizations (optimized)

Without compilation, the remaining time is entirely based on the current processing speed (i.e., the speed of the bytecode interpreter). We track this speed for every worker thread individually by calculating the local tuple processing rate whenever we finish a morsel. The total pipeline duration can then easily be extrapolated based on the remaining tuples in the pipeline, which is always known at that point in time, and the number of active worker threads. We can further refine this extrapolation by using a dynamically growing morsel size, yielding a higher number of sample points.


```

# fig:adaptive: worker function
# n: remaining tuples
# w: active worker threads
extrapolatePipelineDurations(f, n, w):
    r0 = avg(rate in threadRates)
    r1 = r0 * speedup1; c1 = ctime1(f)
    r2 = r0 * speedup2; c2 = ctime2(f)
    t0 = n / r0 / w
    t1 = c1 + max(n - (w - 1) * r0 * c1, 0) / r1 / w
    t2 = c2 + max(n - (w - 1) * r0 * c2, 0) / r2 / w
    switch min(t0, t1, t2):
        case t0: return DoNothing
        case t1: return Unoptimized
        case t2: return Optimized

```

Figure 29: Extrapolation of the pipeline durations.

With compilation, the remaining time of the pipeline also depends on the expected compilation time as well as an estimate of how much faster the compiled code would be. Another aspect that needs to be incorporated is that, while compilation is ongoing, the execution of the pipeline can continue in a multi-threaded setting (using the bytecode or optimized code). We thus have to compute the tuples that can be processed on the remaining threads during the compilation and extrapolate the time needed for the remainder afterwards.

The compilation time of a worker function depends on the generated query plan and is determined empirically in our system. As Figure 28 shows, the number of LLVM instructions of a query correlates very well with its compilation time for all TPC-H and TPC-DS queries. The generated plans contain between 300 and 19,000 instructions for which we observe a near-linear compilation time. For the speed-ups between different execution modes, we use empirical data (see Section 5.5.4). While it is generally difficult to forecast accurate query speed-ups and compilation times, adaptive execution only requires rough extrapolations (see Section 5.5.7).

Figure 29 shows the pseudo code for comparing the different execution modes. In order to reduce the synchronization overhead, the extrapolation is only performed by a single worker thread. We delay the first evaluation by 1 millisecond to increase the accuracy of the estimates and reevaluate after every processed morsel thereafter. After every morsel, the thread computes the average processing speed of all threads and compares the remaining processing time of the execution modes. If the transition to a new execution mode

```

while (true) {
    switch ((++ip)->op) {
        case Op::add_i32: *((int32_t*)(regs + ip->a1)) =
            *((int32_t*)(regs + ip->a2))
            + *((int32_t*)(regs + ip->a3)); break;
        case Op::add_i64: *((int64_t*)(regs + ip->a1)) =
            *((int64_t*)(regs + ip->a2))
            + *((int64_t*)(regs + ip->a3)); break;
        case Op::call_void_i32: (void (*)(int32_t))(ip->lit)
            (*(int32_t*)(regs + ip->a1)); break;
        case Op::call_void_i64: (void (*)(int64_t))(ip->lit)
            (*(int64_t*)(regs + ip->a1)); break;
        ... // around 500 more instructions
    }
}

```

Figure 30: VM code fragment implementing the interpreter loop. `ip` points to the current instruction and `reg` points to the memory storing the registers.

appears beneficial, the thread compiles the worker function and resets all processing rates. This allows one to eventually transition to the fastest execution mode for every pipeline.

5.4 FAST BYTECODE INTERPRETATION

As Figure 24 shows, generating machine code takes a non-trivial amount of time—even without compiler optimizations. An interpreter is therefore a crucial part of our design.

LLVM is a compiler framework that has been designed to generate machine code, but it also contains an interpreter. This interpreter directly executes the LLVM IR without any additional compilation step. Thus, systems that compile to LLVM IR could execute queries using this interpreter. However, as can be seen in Figure 24, the built-in interpreter is extremely slow (over 800 times slower than the corresponding machine code). The reason is that LLVM IR was designed as a versatile and generic format for implementing optimization passes. Its pointer-based in-memory representation allows easy code transformations but is highly cache unfriendly. Furthermore, the execution of an instruction involves a costly runtime dispatch as there is only a single instruction (e.g., integer addition) for all operand widths (e.g., 8, 16, 32, 64 bits).

To make interpretation a viable strategy, we therefore translate the native LLVM IR into an optimized bytecode format for a virtual machine (VM) that can be interpreted much more efficiently. We have to address two key challenges here: First, processing the bytecode should be as cheap as possible in order to minimize the interpretation overhead. Second, an efficient translation into this bytecode has to be possible. The latter is particularly difficult as many standard compiler techniques like liveness analysis have a super-linear worst-case behavior, yielding unacceptable translation times for very large queries. And finally, the VM must behave 100% identical to native machine code as we want to seamlessly switch between interpreted VM code and native machine code. We therefore developed a virtual machine that mostly follows the LLVM instruction set, but aims for cheap interpretation and offers additional functionality for common constructs.

5.4.1 Virtual Machine

Our virtual machine is a *register machine*. When calling an interpreted function, we allocate a register file that holds all values computed during function allocation. This allocation happens on the stack if possible, falling back to heap allocation if the register file is too large. For now, we can pretend that every value computed in the LLVM IR has one fixed position in that register file. As we will see in Section 5.4.3, it is actually undesirable to map values to registers like that, but for now we just assume that all values exist somewhere in the register file. The first two entries in the register file are initialized to 0 and 1, respectively, such that these constants are always readily available in registers.

The instruction set of the VM is *fixed length, statically typed*, and in most places mimics the LLVM IR instruction set. For example, the small LLVM function

```
define i32 @add(i32, i32) {
    %3 = add i32 %1, %0
    ret i32 %3
}
```

will be translated into a very similar VM fragment:

```
add_i32 24 16 20
return_i32 24
```

The *add_i32* instruction loads the two function arguments from the registers 16 and 20 (which are byte offsets into the register file), and writes the result

```

compute liveness and order blocks
for each block b:
  allocate registers for values that become live in b
  for each instruction i in b:
    if i is not subsumed:
      translate i into VM opcodes
  propagate values in  $\phi$  nodes
  release register for values that ended in b

```

Figure 31: Translation of LLVM IR into VM code.

back into register 24. The `return_i32` instruction returns that value to the caller. Note that there is not always a 1:1 correspondence between LLVM instructions and VM instructions, like in this example. First, the LLVM instructions are annotated with types, while the VM instructions have the type baked into to the opcode itself. For example the LLVM `add` is expanded into different add instructions during translation depending upon the argument types. And second, we sometimes collapse multiple LLVM instructions into one VM instruction to handle frequently occurring instruction sequences (see Section 5.4.6).

We use a fixed length encoding for the opcodes to improve the decoding speed. This increases the memory footprint of the translated function relative to native machine code, but it is still much more compact than the original pointer-heavy LLVM IR. As Figure 30 shows, the VM code itself then conceptually consists of a large switch statement³ that evaluates all supported instructions.

In total, the VM handles about 500 instruction/type combinations, each consisting of a single and fairly simple line of C++ in the VM code. The bytecode interpreter code is about 800 lines of code, which is surprisingly small for a component that allows us to interpret arbitrary query plans without modifying the query code generation. This is important for the maintainability of the system as it would be highly unattractive to maintain completely separate code paths for both—native code and interpreted execution.

5.4.2 Translating into VM Code

The translation of LLVM IR code into VM code is shown in Figure 31. It starts by computing the liveness information for register allocation, which is

³ Instead of a single switch statement, we use a dispatch table with *computed goto* statements as that will reduce the branch mispredictions in modern CPUs.

by far the most challenging step of the translation, and therefore discussed below in detail. Afterwards, we know when a value becomes alive within the control flow and when it dies.

With that information, the transformation itself is simple. Note that the transformation exploits the fact the LLVM programs are in Single Static Assignment (SSA) form, i.e., a value is produced exactly once, and never changes during the lifetime of the program.

We iterate over all basic blocks of the program in the order that the liveness computation has determined. For every block, we then check whether values become alive even though the producing instruction is not contained in the block itself (this is rare, but can happen with a complex control flow). If so, we immediately allocate a register for these values. The instructions within the block are then translated into VM opcodes one by one, except for cases where subsequent instructions are *subsumed* by previous instructions, for example when folding a sequence of instructions into one VM opcode (see Section 5.4.6). At the end of the block we copy values into the ϕ nodes of successor blocks if needed (i.e., if the successor block uses ϕ nodes to unify different values in SSA representation), and release registers for all values where the lifetime has ended. Just as the allocation mentioned before, this is also an exception caused by the control flow, as we will discuss below. For the vast majority of cases we allocate registers on demand and release them when the last user of that value is gone. In summary, we consider block boundaries only when the control flow forces us to extend the lifetime of a value.

After this translation step, the VM program is ready for execution. It performs exactly the same work as the native code would, including all function calls and all memory writes, which is important for the switch between interpretation and compilation. There are some engineering details here to make that substitution possible. Calls to interpreted code, for example, need to be patched during the translation to accept an additional parameter (the VM program). However, that is similar to standard compiler techniques for nested functions and does not introduce too much complexity. Our translator has about 2,400 lines of code most of which are dedicated to the register allocation. As the translation operates almost entirely on the well defined LLVM IR language, the additional engineering effort is not too high.

5.4.3 Register Allocation

As mentioned before, there is only one step during the translation into VM code that is algorithmically challenging, and that is the register allocation, i.e., the mapping of LLVM values to register slots. Our problem differs slightly from traditional register allocation, as we only use virtual registers and therefore could allocate a (nearly) arbitrarily large number of them. However, we clearly do not want to do this: The register file is accessed very frequently during interpretation, and therefore should always be in the L1 cache. A large register file wastes precious L1 cache entries.

Our register allocation problem is therefore the following:

1. Assign a register slot to every LLVM value in the program
2. Make sure that a register is only shared between different values if their lifetimes do not overlap
3. Minimize the total number of registers
4. Translate very large programs efficiently

In principle, register allocation is a well understood problem in compiler construction [93]. In order to do register allocation we need liveness information, i.e., we have to know for each basic block which values are alive and which are dead. However, computing this liveness information has super-linear runtime in the number of basic blocks, which can make these algorithms prohibitively expensive for large functions. And unfortunately, some of our queries do compile into very large functions with thousands of basic blocks and tens of thousands of values. This is very different from handwritten programs, which tend to consist of small functions. Register allocators try to avoid the expensive liveness computation by splitting the life-ranges via spilling to memory [96]. But this is not really an option for us (in contrast to regular machine code), as we would then have to find a mechanism to minimize the spill region, as that would have to be cache-resident, too. Some JIT systems therefore restrict the register allocation to values within a single basic block (which is easy) or consider only a fixed number of neighboring basic blocks. This approach is computationally simple but can lead to a poor register allocation.

We developed a new linear-time register algorithm that recognizes and utilizes loop structures to quickly approximate the optimal register allocation. Finding the optimal register allocation for a program in SSA form with an unbounded number of registers is super-linear. Instead, our register allocator may

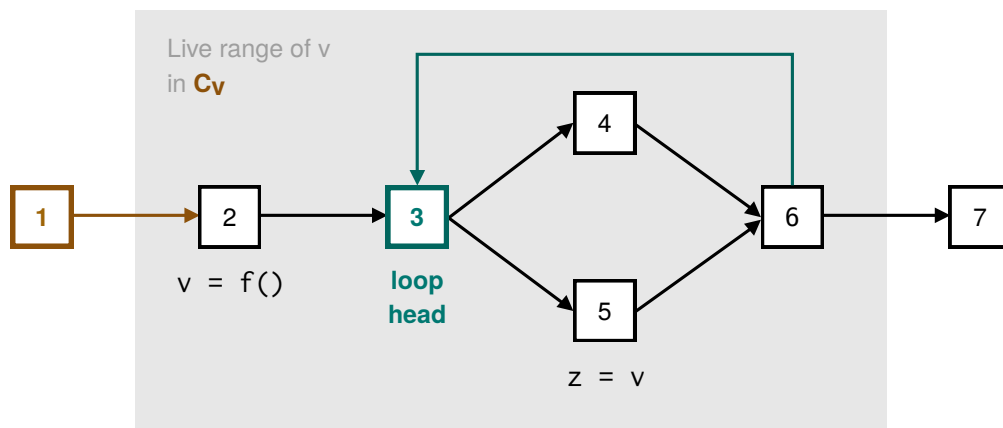


Figure 32: Computing the liveness of a variable x . The vertices are basic blocks, which are connected by control flow edges (i.e., branch instructions).

sometimes needlessly extend the lifetime of a variable within the bounds of the innermost loop that contains all uses of that value. In practice, however, this only occurs with complex control flows, has imperceptible effects and serves as reasonable trade off for the linear worst-case behavior.

Having a linear runtime algorithm is very important for the adaptive execution framework. As we will see in Section 5.5.5, the regular LLVM compiler is de facto unable to compile some very complicated queries due to the super-linear algorithms used. Indeed, we have encountered machine-generated queries where the largest function consists of 300,000 values and thousands of basic blocks. An algorithm with super-linear runtime for such functions thus leads to unacceptable compile times (hours or even days).

To give an impression of different register allocation strategies, we report the size of the register file for different allocation strategies for the relatively large TPC-DS query 55: If we just allocate values to registers without reuse we need 36 KB, which is larger than our L1 cache. Using a greedy assignment strategy instead where we consider a fixed window of basic blocks for the lifetime, we need 21 KB. This is better and sufficient for some JIT compilers, but still quite large. The algorithm that we present below reduces this number to 6 KB, which is much more reasonable.

5.4.4 Linear-Time Liveness Computation

Our algorithm is based upon two key concepts: 1) we compute that liveness of a value as a live-range with a start block and an end block. The traditional

```

// compute the liveness of values in function F
ComputeLiveness(F):
  // find loop structures in F
  label all basic blocks in F in reverse postorder
  compute the dominator tree D for each basic block
  label all nodes in D with pre-/postorder numbers
  mark the first basic block in F as loop head
  for each jump edge  $j: B \rightarrow B'$ :
    if  $B'$  is ancestor of  $B$  in  $D$ :
      mark  $B'$  as loop head
  for each basic block  $B$ :
    associate  $B$  with the next dominating loop head
  for each loop:
    compute the first and last block of the loop
    compute the next dominating loop head
    label loop with nesting depth
  // use the loop information to compute lifetimes
  for each value  $v$  in  $F$ 
     $B_v$  = set of basic blocks containing
      definition and users of  $v$ 
     $C_v$  = innermost loop containing all blocks in  $B_v$ 
     $L_v$  = empty lifetime interval
  for each  $B$  in  $B_v$ :
    if  $C_v$  is innermost loop for  $B$ :
      extend  $L_v$  with  $B$ 
    else:
      extend  $L_v$  with outermost loop below  $C_v$ 
        that contains  $B$ 

```

Figure 33: Linear-time algorithm for liveness computation.

method of computing the liveness for each block individually inherently has $O(n^2)$ runtime. And 2) we keep the live-range of each value as tight as possible by labeling the blocks according to the control flow and by explicitly handling loops. This is illustrated in Figure 32: The basic blocks in this figure are labeled in *reverse postorder*, which matches the control flow order. The value x is created in block 2 and consumed in block 5. Naively one could think that the lifetime of x is therefore the interval $[2,5]$, but this is incorrect: Block 5 is part of a loop that starts in 3 and which involves the blocks $[3,6]$. Any of these blocks can reach block 5. Therefore, we extend the lifetime to include the *containing loop* of the reader, which results in the life range in $[2,6]$.

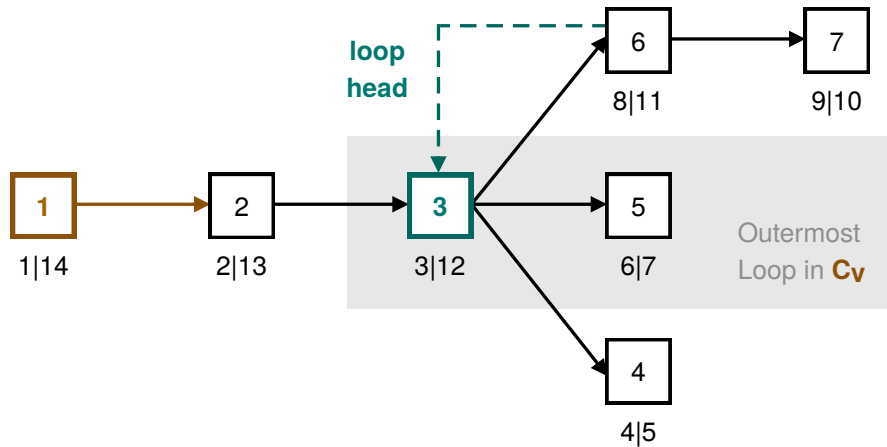


Figure 34: Dominator tree annotated with pre-/post-order.

The full algorithm is shown in Figure 33. It operates in two phases: In the first phase, it identifies all loops that occur in the function and associates each basic block with the innermost enclosing loop⁴. With this information we can compute the lifetime of a value by identifying all basic blocks that contain definition or uses of a value and lifting these blocks to the level of the innermost loop that contains all blocks. Conceptually, a value is alive from its definition to its last user, including all blocks that might be traversed along the way due to loop constructs.

We now look at the algorithm in more detail. It starts by labeling (and ordering) all basic blocks in *reverse postorder*, i.e., a block is placed after all its incoming blocks. Ignoring loops this directly corresponds to the control flow order, and for a human would be the “natural” way to order the blocks in a programming language. This order is required for the next algorithm step and has the added advantage of making sure that the block labels are meaningful regarding the control flow. Using this labeling, we can compute the dominator tree D efficiently [41, 36], which, for each basic block, tells us the closest basic block that must have been executed before. For lookup purposes we label all nodes in D with pre-/post-order numbers [44]. This labeling allows us to determine ancestor/descendant relationships in $O(1)$. The dominator tree for our running example is shown in Figure 34. Using the pre/postorder numbers we can immediately see that, e.g., block 2 transitively dominates block 6, as the interval $[8,11]$ of block 6 is contained in the interval $[2,13]$ of block 2.

All this infrastructure is used to identify loops. To avoid edge cases for blocks outside of loops, we pretend that the whole function body is part of one

⁴ To avoid edge cases for blocks outside loops the algorithm behaves as if the whole function body is contained in one huge outermost loop

large loop, and we mark the first block of the function as the *loop head* (i.e., the entry point of the loop). Now we look at all jumps between pairs of blocks B and B' . If B' is an ancestor of B in the dominator tree D , we have found a loop, and we mark B' as the loop head. In our example block 6 jumps to block 3, which dominates 6, and thus block 3 is a loop head, i.e., the entry point of a loop. After identifying all loops, we associate each block with their innermost containing loop, represented by the nearest dominating loop head. We use a disjoint set data structure with *path compression* here to make this computation fast. We remember the first and the last block of a loop (according to the block labels) and the loop in which it is nested. In our example the loop starting at block 3 contains the block 3–6, and is contained in the top-level (pseudo) loop starting at block 1. Finally, we compute the nesting depth for each loop.

While this computation is involved and uses several non-trivial algorithms, the overall complexity of each step is linear. Indeed, most of the complexity stems from the fact that we want to guarantee linear runtime: We could, for example, leave out the pre/post-order labeling or the path-compression, but we would then get super-linear runtime in subsequent steps. The same is true for the choice of the dominator tree algorithms.

Using this loop information, the liveness computation for each individual value v becomes simple. We identify the set U_v of all blocks that contain either the definition or the uses of v . If the containing loop was the same for all these blocks, the lifetime would simply be the span from the first block to the last block, according to the reverse postorder labeling. In the general case, we identify the least common loop C_v that contains all blocks from U_v . We extend the lifetime of v to include the blocks from U_v within C_v that are not located within nested loops. For every other block b in U_v , we extend the lifetime of v to include all blocks of the outermost loop within C_v containing b . In our example the containing loop C_v for value v is the whole function, the definition of v in block 2 is immediately in that loop, but the use of v in block 5 is one loop level deeper. As a result, the lifetime of v is the interval $[2,6]$. This whole computation is very cheap due to lookup structures we have prepared while analyzing the loops in the first phase.

Note that some care is required for LLVM's ϕ nodes: The ϕ nodes are used for the Single-Static-Assignment (SSA) form, and they pick a value depending upon the incoming edge that has led to the basic block with the ϕ node. For the purpose of lifetime computations, the arguments of ϕ are “read” at end of the corresponding incoming block, and the ϕ node is “written” immediately afterwards in the same block, and then “read” in the block that contains the ϕ

node. This is not particularly difficult to implement, but one has to keep that in mind when computing the liveness for ϕ nodes.

5.4.5 Interoperability

We interpret the original LLVM IR using our virtual machine. Therefore, our bytecode interpreter behaves equivalently to generated machine code (except for speed differences, of course). This is important, because it allows us to seamlessly switch between interpretation and machine code, without modifying the rest of the system.

The interoperability between bytecode and machine code, however, raises a problem. While a function pointer suffices to run machine code, we need to interpret the bytecode with the virtual machine. So instead of a direct function call we need to call additional dispatch code (see Figure 27) and pass to it the function's bytecode as an additional argument. We could then differentiate both signatures by tagging the pointer and dynamically call the respective function, but that would be quite invasive and would introduce unnecessary branches. Instead, we always pass an extra pointer argument to the function even though it is redundant in the machine code case. This allows us to transparently switch from interpreted to compiled code by replacing the function pointer and inject the additional argument.

The reverse direction is simpler as we can call existing C++ code from both, generated machine code and from our VM. We just have to make sure that a suitable call instruction is available in our VM for every existing function signature. Referring to Figure 30, the opcode `Op::call_void_i32` is required to call C++ functions with a single 32 bit integer parameter and no return value. As we know all exported C++ functions, we can identify missing opcodes at compile time.

5.4.6 Optimizations

While being possible, it is sometimes inadvisable to translate LLVM instructions independent from each other. One example for this is overflow checking. Any arithmetic that occurs within a query is checked for overflows in order to report overflow errors to the user. With LLVM, this check boils down to 4 instructions that are always executed in sequence. With the bytecode interpreter, our translator recognizes this sequence, and replaces it with a single VM byte-

code that performs all four steps at once. This greatly reduces the number of instructions for some queries and decreases their execution time.

Another frequently occurring pattern is the `GetElementPtr` (i.e., pointer arithmetic) instruction followed by a load or store. These sequences are also recognized during the translation and merged into one VM opcode to reduce the instruction count.

In general, it would make sense to translate a large corpus of queries, and to check for frequently occurring sequences of instructions in order to replace them by macro instructions. One candidate for that could, for example, be NULL handling, which also tends to create similar instruction sequences. In future work, we will expand this mechanism to recognize more of these constructs.

5.5 EVALUATION

In this section, we experimentally compare the adaptive query execution framework discussed in Section 5.3 with different statically chosen execution modes. We also devote special attention on the bytecode interpreter introduced in Section 5.4 to answer the question whether query interpretation adds additional value to compilation-based databases.

Our experiments are performed in HyPer, a database system that directly generates LLVM IR, and, so far, always compiled it to machine code. By default, *optimized* compilation was used, which enables all machine-specific (backend) optimizations after executing a number of hand-picked LLVM IR optimization passes (peephole optimizations, reassociate expressions, common subexpression elimination, control flow graph simplification, aggressive dead code elimination). We also implemented an *unoptimized* compilation mode, which also generates machine code but disables most compiler optimizations to improve compile times. Specifically, this mode enables fast instruction selection, does not execute any LLVM IR optimization passes, and uses a low backend optimization level. Our *interpreter* translates the LLVM IR directly into the bytecode discussed in Section 5.4. Finally, the *adaptive* execution mode interleaves machine code generation and execution as described in Section 5.3.

Unless otherwise noted, the experiments have been performed on a desktop system with an 8 core AMD Ryzen 7 1700X CPU, 32 GB of RAM, LLVM 3.8, and Linux 4.11.

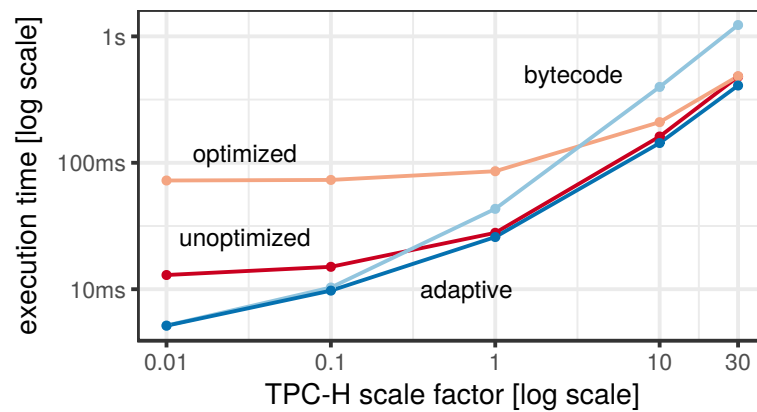


Figure 35: Geometric mean of all TPC-H queries including planning, compilation, and execution using 8 threads for different scale factors and execution modes.

5.5.1 Static vs. Adaptive Mode Selection

Let us first investigate whether adaptive switching of the execution mode can compete with a static up-front decision. In this experiment, we run all 22 TPC-H queries on scale factors ranging from 0.01 (around 10 MB) to 30 (around 30 GB).

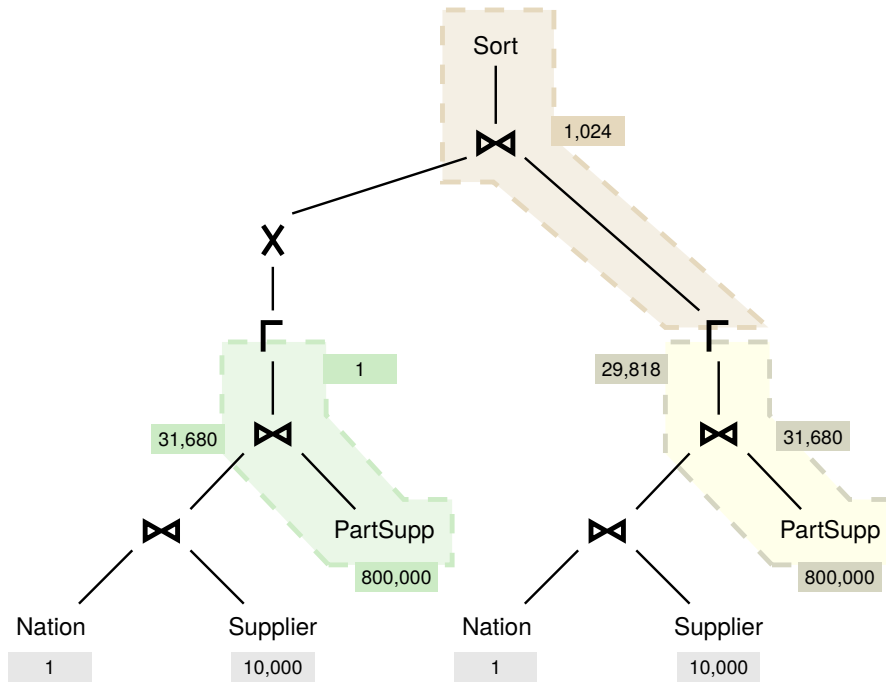
Figure 35 presents the geometric means over all queries and for all execution modes. Without having prior knowledge about the exact data size, adaptive execution is able to always compete with the best statically chosen execution mode. For the scale factors 0.01 and 0.1, the superior strategy is determined solely by the query latency which clearly favors interpretation over compilation. At these data sizes, adaptive execution never chooses to compile and performs just as well as pure bytecode interpretation. Starting from scale factor 1, it becomes viable to compile many of the pipelines, making unoptimized compilation competitive. However, adaptive execution is still able to outperform unoptimized compilation as fast pipelines can still be processed as bytecode. Finally, at scale factor 30 the queries run long enough to justify the optimized compilation. Adaptive execution now picks the best out of three execution modes per pipeline and outperforms both compilation modes noticeably. At even larger scale factors, we expect this trend to continue, with optimized compilation becoming the main competitor for adaptive execution. However, we also expect that adaptive execution will continue to have the overall lowest processing time as there will still be cheap pipelines in the query plans, that can be executed immediately.

5.5.2 Adaptive Execution in Action

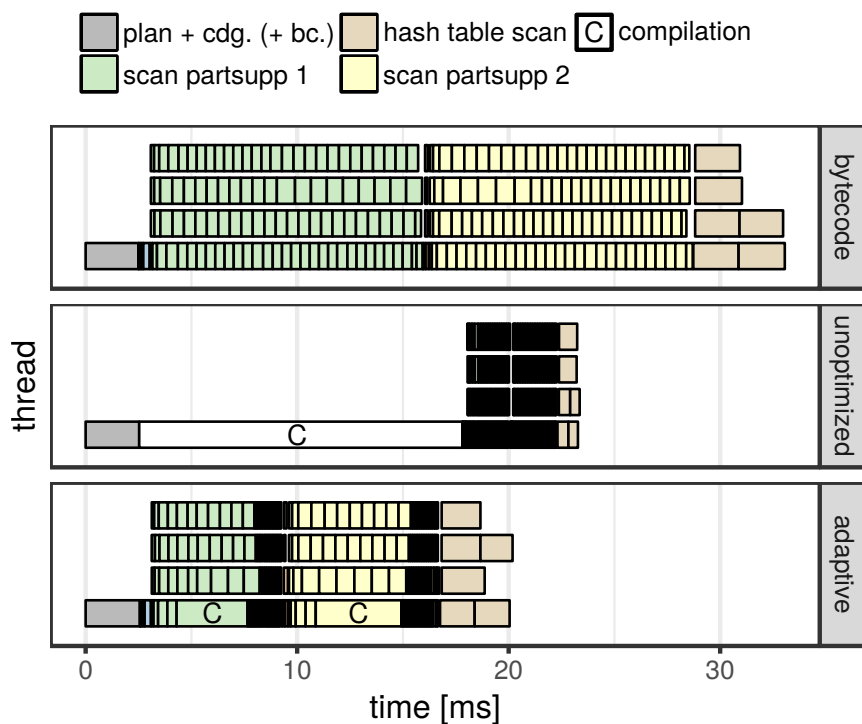
In a next step, we investigate the adaptive behavior of the framework based on TPC-H query 11 on scale factor 1 using 4 threads⁵. We compare adaptive execution with its competitors using Figure 36, which shows the query plan in (a) and a dynamic execution trace with precise timing information about the processed morsels in (b). Starting with the bytecode interpreter, the figure shows that the database quickly uses all 4 worker threads to process the pipeline morsels in parallel. It also reveals that the amount of work is distributed very unequally among the 7 pipelines and that most of the time is spent on the processing of the pipelines “scan partsupp 1” and “scan partsupp 2”. Unoptimized compilation, in contrast, uses a significant proportion of the time for the initial single-threaded compilation of the query plan. Afterwards, the morsels can hardly be distinguished from each other as the processing with compiled pipelines is much faster. The execution trace of optimized compilation looks very similar to the one from unoptimized compilation but is not shown for graphical reasons, as the additional compiler optimizations lengthen the compilation time to 103 milliseconds. In summary, unoptimized compilation dominates the other statically chosen execution modes for this query (at scale factor 1) due to being a good tradeoff between an increased efficiency and a fast query preparation. These observations already indicate, that the quality of a static up-front decision highly depends on the complexity of individual pipelines and the data that is being processed.

Figure 36 also shows the execution trace of our adaptive execution mode which is able to outperform all of its competitors. Very similar to the pure bytecode interpreter, adaptive execution can immediately start to process the pipeline morsels on all 4 worker threads. After 1 millisecond, it determines for the two largest pipelines that switching the execution mode is worthwhile and therefore dedicates a worker thread to compile them. As the compilation is restricted to a single function, it only takes a fraction of the time we observed when transforming the whole query plan. Once compiled, all worker threads automatically shift gear to the newly-created machine code and process the remaining morsels very efficiently. However, the unequal complexity distribution favors the compilation of only 2 out of 7 pipelines. Thus our framework processes the remaining pipelines using the bytecode interpreter and finishes the query 10%, 40% and 80% faster than the competitors (i.e., unoptimized compilation, bytecode interpreter and optimized compilation).

⁵ Query 11 has been chosen such that the individual morsels are graphically distinguishable.



(a) Plan



(b) Execution trace

Figure 36: Plan and execution trace of TPC-H query 11 on scale factor 1 using 4 threads. The optimized mode is not shown, as its compilation takes very long (103ms).

Table 4: Planning and compilation times in ms for TPC-H queries on PostgreSQL (“PG”), MonetDB (“Monet”), and HyPer.

TPC-H #	plan		HyPer				
	PG	Monet	plan	cdg.	bc.	unopt.	opt.
1	0.1	0.8	0.2	0.7	0.4	6	42
2	1.0	0.7	0.7	1.5	1.2	23	149
3	0.3	0.5	0.4	0.9	0.7	10	69
4	0.2	0.4	0.2	0.7	0.4	7	47
5	1.2	0.8	0.7	1.2	0.9	15	104
6	0.1	0.5	0.2	0.5	0.2	3	15
7	0.9	0.5	0.6	1.2	0.9	16	108
8	1.3	1.0	0.8	1.4	1.1	19	130
9	1.9	0.6	0.7	1.2	0.9	16	109
10	0.5	0.9	0.5	1.0	0.7	12	81
max	1.9	1.0	0.8	1.5	1.2	23	149

5.5.3 Planning and Compilation Time

Bytecode interpretation is a viable approach to provide a low-latency execution mode in compilation-based databases. In order to provide evidence for this statement, we evaluate the planning and compilation times of HyPer and compare them with PostgreSQL 9.6, which uses Volcano-style interpretation, and MonetDB 1.7, which uses column-at-a-time processing. Table 4 shows the planning times for TPC-H queries 1 through 5 and the maximum over all 22 queries. For TPC-H, plan generation (labeled as “plan” in the table), which includes parsing, semantic analysis, and query optimization, is very fast in all systems. While MonetDB and PostgreSQL can directly execute this plan, HyPer generates LLVM IR code in the code generation phase (abbreviated as “cdg.” in the table). LLVM IR generation typically takes slightly longer than planning, but is still very fast (less than 2ms over all 22 TPC-H queries). The next phase in HyPer is either bytecode (“bc.”), unoptimized (“unopt.”), or optimized (“opt.”) machine code generation. The table shows, that even unoptimized machine code compilation is generally around 10x slower than planning and code generation. Optimized compilation is even slower and takes up to 150 ms for TPC-H. Bytecode generation, on the other hand, is very fast and is always finished in less than 2 ms.

Table 5: Execution times of TPC-H queries on scale factor 1 on PostgreSQL (“PG”), MonetDB (“Monet”) and HyPer. The geometric means (“geo.m.”) are over all 22 queries.

TPC-H #	1 thread					8 threads		
	PG	Monet	bc.	unopt.	opt.	bc.	unopt.	opt.
1	4908	484	858	161	77	170	34	16
2	254	5	94	13	8	25	5	3
3	1258	64	323	104	80	54	21	17
4	193	56	352	67	45	57	16	12
5	516	51	362	60	37	67	14	10
6	102	38	18	23	27	5	4	5
7	664	71	252	79	61	52	16	14
8	1514	46	461	56	40	67	14	12
9	2345	139	863	183	135	123	35	35
10	1138	46	167	57	51	35	14	12
geo.m.	497	57	232	60	46	45	15	12

5.5.4 Performance of Interpreted and Compiled Code

Let us next compare the execution times of the bytecode interpreter and the compiled machine code. Table 5 shows the TPC-H performance on scale factor 1 for the different execution modes and compares them with MonetDB as well as PostgreSQL. Considering the geometric mean across all 22 queries, the bytecode interpreter is 3.6 times slower than unoptimized machine code and 5.0 times slower than optimized machine code. While interpreted code is slower than compiled code, it is still 2.1 times faster than PostgreSQL and scales just as well as compiled code when multiple cores are used. The bytecode interpreter is, however, slower than MonetDB, which uses combining pre-compiled, per-column primitives. This highlights the fact that even a fast bytecode interpreter is not capable of achieving performance competitive with compilation.

In order to better understand the performance results of Table 5, we measured some important⁶ CPU counters⁷ for TPC-H queries 1 and 5. These two queries are quite different: query 1 is very computation-heavy, while query 5 is dominated by join processing. As most queries spent the vast majority of their

⁶ Instructions cache misses, which can be problem in some systems [122, 121, 115], are not shown as they are negligible in our execution engine (even in the interpreter).

⁷ These counters were measured with perf on an Intel Haswell CPU, because on our recent AMD CPU perf does not yet support all counters.

Table 6: CPU counters ($\times 10^6$) for TPC-H queries 1 and 5 on scale factor 1 using 1 thread.

	TPC-H query 1			TPC-H query 5		
	bc.	uno.	opt.	bc.	uno.	opt.
cycles	2669	606	306	1500	360	170
instructions	6257	1530	829	2700	510	200
IPC	2.3	2.5	2.7	1.8	1.4	1.2
branches	548	165	137	280	56	20
branch misses	1.4	1.2	0.7	4	3.2	3.5
L1 accesses	3993	476	253	1800	130	60
L1 misses	14	13	12	6	6	5
LLC accesses	1.8	1.7	1.7	4	5	4
LLC misses	0.6	0.6	0.6	2	2	1.2

time in joins, query 5 is fairly representative for the rest of TPC-H, and query 1 is a bit of an outlier. As presented in Table 6, the counters for the unoptimized code generally lie between the interpreted code and the optimized code. In the following, we therefore mostly contrast interpretation and the optimized code.

Maybe the most striking feature of the bytecode interpreter is that it executes 7.5 times (query 1) and 13 times (query 5) as many instructions as the optimized machine code. This sheer number of instructions executed by the interpreter explains most of the performance difference in comparison with machine code. Even simple operations like 64-bit integer addition, result in around 10 machine instructions with bytecode. Another important difference is the number of level 1 data cache accesses, which are 13.5 times (query 1) and 30 times (query 5) higher in the interpreter than in the optimized code. The high number of L1 accesses are due to the registers of the interpreter being stored in cache (as opposed to CPU register). Most other counters including the L1 data misses and last level cache (LLC) misses are fairly similar in the different execution modes. This shows that the bytecode and registers are largely cache resident, despite our fairly large instruction set and our frequent register accesses. Most cache misses are due to unavoidable data access during query processing (e.g., for probing hash tables). Finally, one can observe that in query 5, which is more representative of the rest of TPC-H, the optimized machine code actually has a lower instructions per cycle (IPC) metric than the bytecode and the unoptimized machine code. This shows that a higher IPC is not necessarily better, as optimized, lean code has more CPU stalls due to unavoidable data access.

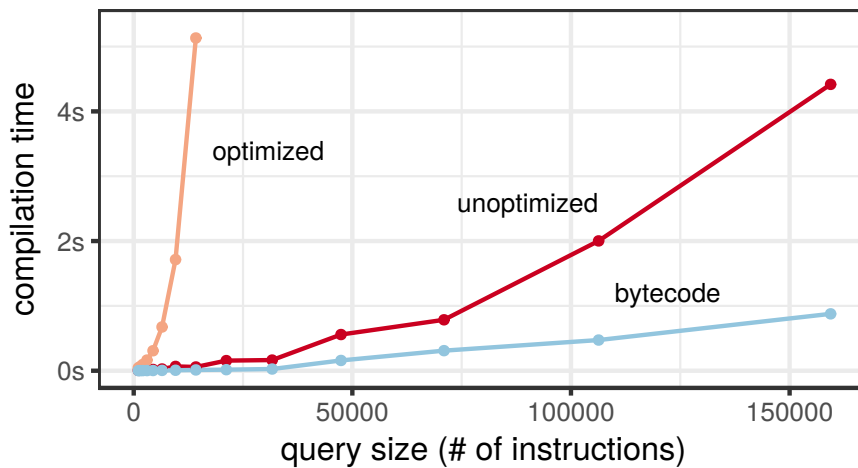


Figure 37: Compilation times of queries with a large number of instruction using optimized compilation, unoptimized compilation and interpretation.

5.5.5 Compiling Very Large Queries

In Section 5.3, we introduced a linear cost function that estimates the compilation time based on the number of instructions in the pipeline. However, we derived this function from the TPC-H and TPC-DS benchmarks which do not contain particularly complex queries. Machine-generated queries, on the other hand, can easily comprise multiple MB of SQL text with very unpleasant properties for the query compiler. In our last experiment we therefore investigate the effects of very large queries on our three execution modes and show that fast translation into bytecode is indispensable for these workloads.

Our sample queries consist of a single table scan and an increasing number of aggregate expressions. By scaling this number from 10 to 1900, we receive query plans that contain between 1,000 and 160,000 LLVM instructions, most of which are in a single large function. Figure 37 shows the compilation times of these queries with the different execution modes. Above all, the measurements show that optimized LLVM compilation is no longer a viable approach for larger query sizes. Its compilation times are characterized by an explosive growth and exceed the 4 seconds mark already for 10,000 LLVM instructions. Without optimization passes, the query compilation scales better but still requires 4.4 seconds for the largest of our queries. In comparison, the bytecode interpreter scales perfectly and is able to process this very large query in only 0.9 seconds. This workload stresses the importance of the fast bytecode translation that we introduced in Section 5.4. The translation allows us to execute

queries of (almost) arbitrary size with the interpreter and adaptively compile parts of the query whenever efficiency is needed.

5.5.6 Adaptivity to Data Size and Parallelism

We now investigate how varying data sizes and different degrees of parallelism affect adaptive execution. We chose TPC-H query 4 as an example and analyze the query execution on the scale factors 0.25 and 1 using 1, 2, and 4 threads. Figure 38 shows both the plan of the query and the resulting 6 execution traces. Similar to Figure 36, the execution traces show the timing information of the morsels that are being processed as well as the pipelines to which they belong. The morsels with the labels C/C* indicate compilation tasks without/with compiler optimizations.

It is immediately striking that the execution traces differ significantly depending on the data size and the thread count. On scale factor 0.25, we can observe that the scan of the `Orders` relation is always executed with the interpreter whereas the scan of the `LineItem` relation is always getting compiled. On scale factor 1, in contrast, the scan of the `Orders` relation is getting compiled with 1 and 2 threads but still uses the interpreter with 4 threads. Furthermore, the compilation of the `LineItem` scan even uses optimization passes on scale factor 1 for 1 and 2 threads. The final hash table scan is, independent from the scale factor, always executed with the interpreter.

These observations highlight two important characteristics of adaptive query execution. First, the pipelines are getting compiled more often and more aggressively when the number of threads is *smaller*. This is because the costs of a single-threaded compilation, even with enabled optimization passes, can amortize through the higher amount of work that is left once the compilation finishes. Or, put another way, single-threaded compilation is less attractive with a higher number of threads as the interpreter will already process most of the pipeline during compilation. Second, the pipelines are getting compiled more often with a *larger* data size. This is no surprise as a larger amount of data usually favors a higher processing efficiency. The experiment shows the importance of the decisions at pipeline-granularity as the amount of work might be often small for some of the pipelines.

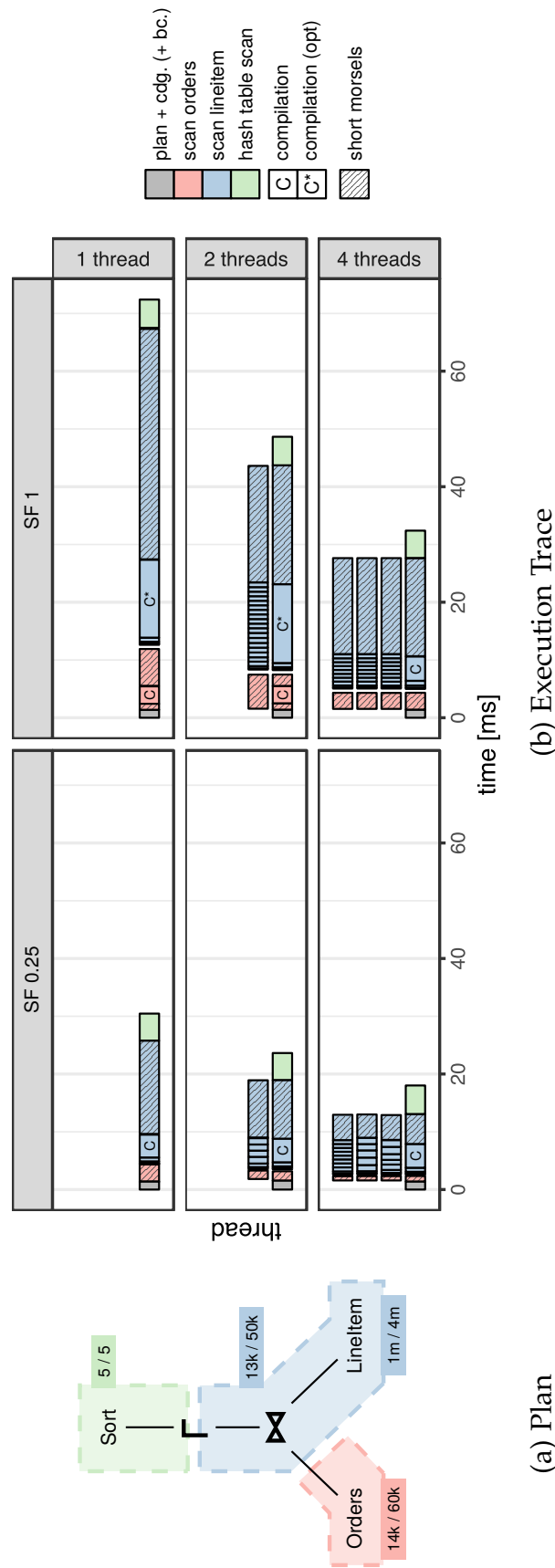


Figure 38: Plan and execution traces of TPC-H query 4 for adaptive execution on scale factors 0.25 and 1 using 1, 2, and 4 threads.

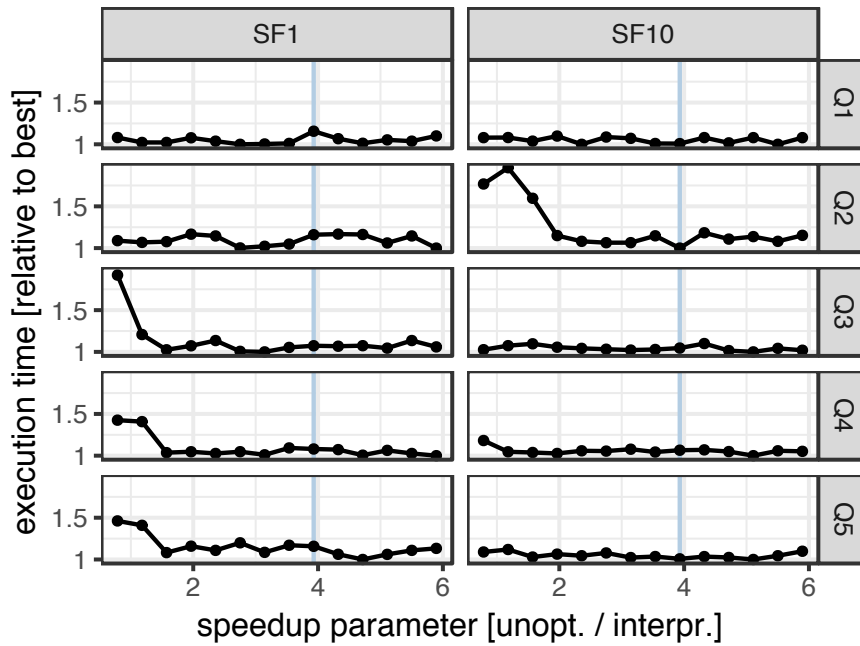


Figure 39: The effect of different speed-up factors between the bytecode interpreter and unoptimized compiled code on the execution time for the TPC-H queries 1 – 5 at the scale factors 1 and 10 using 8 threads. The constant used in our system is marked in blue.

5.5.7 Resistance to Estimation Errors

We mention in Section 5.3 that adaptive execution requires a rough estimate of how much faster code will be at different compilation levels. For that purpose, we introduced speed-up factors that were determined empirically in our system. We now want to illustrate that these constants do not need to be carefully calibrated as adaptive execution is resistant to estimation errors.

Figure 39 shows the effect of different speed-up factors between the bytecode interpreter and unoptimized compiled code on the execution time of the TPC-H queries 1 – 5 at scale factors 1 and 10. We further assume that compiler optimizations increase the speed-up of compiled code by around 30 percent. Despite these arbitrary constants, we can see that the execution times of these queries stay roughly the same. The reason for that is that adaptive execution evaluates the execution modes in a “careful” way.

If the speed-up is assumed to be much larger than the actual speed-up, one would compile more pipelines than required. In the experiment, however, we do not observe a significantly longer execution time for higher speed-up factors as we usually delay the first extrapolation of the pipeline duration in favor of more accurate throughput estimates. This allows many small pipelines to

already complete the execution with the bytecode interpreter even though the compiled code would appear to be arbitrarily faster. Furthermore, in a multi-threaded setting, we can continue to process the query with the bytecode interpreter and abort ongoing compilations if the processing finishes early.

If the speed-up is, on the other hand, assumed to be much smaller than the actual speed-up, one would prefer interpretation more often. This effect is, in fact, observable but only for extreme speed-up values that are close to 1. Intuitively, if the system assumes that there is almost no difference between the interpreted and the compiled execution mode it is worthwhile to fully process the query with the bytecode interpreter.

For all speed-up values in between these two extremes, the decisions are usually unchanged for the pipelines which results in almost the same query execution times. This experiment therefore shows that adaptive execution does not depend on carefully calibrated constants but is very robust.

5.6 RELATED WORK

Many papers on compilation-based query processing only report execution times without stating the time it takes to generate the machine code itself. Those papers that do report them, show compilation times between 5ms and 37ms when LLVM IR is used as a target language [63] and closer to 1 second for compilation to C [59, 111]. As compilation is becoming widespread, we expect that compilation times will receive more attention as any industrial-strength system must deal with very large queries. Indeed, our personal experience has been that after transitioning from standard benchmark queries, which are usually well-designed and “sane”, to real-world customer queries, which are sometimes very “interesting”, query compilation latency becomes a major problem [125]. We therefore believe that adaptive execution is a crucial component for making query compilation truly practical—in particular since traditional engines and modern columns stores (e.g., [70, 1, 101, 91]) do not have large compilation times. In the following, we describe how adaptive compilation can be integrated into other systems, and discuss other approaches for reducing compilation time.

Adaptive execution has been designed for systems that directly compile queries to LLVM IR, which includes HyPer [87] and Peloton [82]. MemSQL is another system that is based on compilation that would benefit from adap-

tive execution. It originally compiled queries to template-heavy C++, which resulted in very high compilation times. Likely for this reason, recent versions compile to a high-level imperative language called MemSQL Plan Language, which is then lowered down via a mid-level intermediate language called MemSQL Bit Code to LLVM IR [90]. Since the MemSQL Bit Code can be interpreted, switching between interpretation and execution could easily be implemented at that level. A similar approach like adaptive execution could also be applied to systems like LegoBase [59] and its successor systems [111, 118], both of which can either execute queries through the Java VM or by compiling to a low-level language like C. Adaptive execution might also be useful for traditional (e.g., Volcano-style) systems that use compilation to specialize the query engine code for a particular query [141, 140, 124]. Microsoft Hekaton, which is part of SQL Server, compiles stored procedures to C [37]. For this use case, compilation times are arguably less important than for ad hoc queries because stored procedures are generally defined infrequently, but executed often.

Automatic plan caching, i.e., reusing query plans between subsequent executions of the same (or a similar) query, is another, orthogonal approach for reducing compilation times. However, plan caching, like explicit prepared statements, cannot hide the compilation time of the first incoming query. For interactive applications this means that the initial user experience of compilation-based systems is far from ideal. Another disadvantage of plan caching is that recurring queries are often not exactly the same, but, e.g., differ by the selection constants. Our adaptive approach can re-optimize queries on every execution, which has the advantage that the specific query constants are visible to the query optimizer, potentially leading to better query plans. Nevertheless, it would also be possible to combine our approach with plan caching. Indeed, one could extend adaptive execution to incorporate multiple executions of the same query by keeping track of how often each pipeline is executed. In this design, eventually all pipelines of frequently-executed queries would be compiled with optimizations.

Adaptive execution bears similarities with the execution engines of modern managed languages like Java (HotSpot), C# (CLR), and JavaScript (V8, JägerMonkey). These systems initially execute code in an interpreter and then, for hot code, dynamically switch to compilation. Our adaptive execution framework can be considered a database-specific implementation of similar ideas. However, for maximum performance, database systems require precise control over memory management and are therefore generally written in (or generate) low-level languages. Therefore, databases cannot use automatic solutions at

the language level, which, to the best of our knowledge, only exist for managed languages. On the other hand, in contrast to a general-purpose programming language, a database system knows much more about the code structure and the instructions generated. This simplifies the design and implementation of adaptive execution (e.g., we do not implement LLVM IR instructions that we do not generate) and allows database-specific optimizations (e.g., macro operations for common operations like overflow checking).

5.7 SUMMARY

We showed that interpretation and compilation are both important building blocks for achieving low query latency and high throughput. We also presented an adaptive execution framework that dynamically and automatically adjusts the execution mode of a query to minimize its overall execution time. In this approach, all decisions are made at a pipeline granularity and are based on runtime feedback instead of having to decide up-front. We further proposed a bytecode interpreter that features a linear-time translation of LLVM IR into efficient bytecode. Using this interpreter and the existing LLVM compiler with optional optimization passes, our system was able to dynamically adapt to data sizes ranging from 10MB to 30GB and outperform all statically chosen execution modes for queries in the TPC-H benchmark.

We described the implementation details of adaptive execution in the context of HyPer. We want to point out, however, that HyPer merely serves as a basis for evaluation and that its design principles are no strict prerequisite for the adaptive query execution. The two most important requirements of adaptive execution are the compatibility of the generated language to virtual machine bytecode and sufficiently fine granular control over the query execution. Within HyPer, the generated LLVM code, and its SSA form in particular, simplify the translation into a bytecode for a custom virtual machine significantly. Other intermediate languages are still able to execute queries adaptively as long as they can be paired with a virtual machine featuring a fast bytecode translation and a good interoperability with the compiled code. In addition, fine-granular control is required to be able to efficiently switch the execution mode and track the query progress. The processing of data in small chunks, as done with morsel-driven parallelism, is not specific to HyPer but can be found in different variations in modern execution engines like IBM

BLU [101] and Quickstep [91]. For these reasons, we believe that adaptive execution is directly applicable to other compiling query engines—including ones that do not use LLVM for code generation.

6 | ELIMINATING LATENCY WITH WEBASSEMBLY

Excerpts of this chapter have been published in [67].

6.1 INTRODUCTION

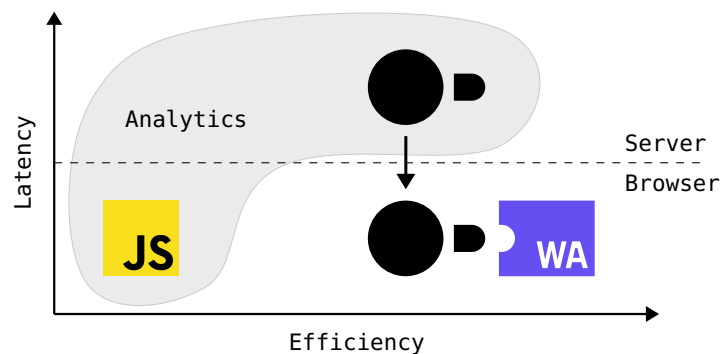


Figure 40: Browser-based analytics tools process data either locally with a low efficiency or on servers with a high latency. DuckDB-Wasm pushes the boundaries with fast analytical processing for the Web.

The web browser has evolved to a universal computation platform. Its rise has been accompanied by increasing requirements for the browser programming language JavaScript. JavaScript was designed to be flexible which comes at the cost of a reduced processing efficiency. This is pronounced when considering the execution times of complex data analysis tasks that often fall behind the native execution by orders of magnitude. In the past, analysis tasks have therefore been pushed to servers that tie any client-side processing to additional round-trips over the internet. These round-trips introduce network latencies that negatively affect interactive data exploration [78].

The processing capabilities of browsers were boosted significantly in 2017 with the release of WebAssembly [46]. WebAssembly is a collaborative effort to design a portable low-level binary instruction format for a safe stack-based virtual machine. It is supported by major browser engines today and serves as efficient compilation target for programming languages like C++. WebAssem-

bly aims to execute programs at native speed and supersedes JavaScript for performance-critical applications in browsers.

The rise of WebAssembly presents an opportunity for the database DuckDB to bring fast analytical data processing to the Web. DuckDB is a purpose-built *embeddable* database for interactive analytics [100, 99]. Embeddable databases are linked to programs as libraries and run in their processes. This design distinguishes DuckDB from stand-alone data management systems and allows for tight integrations into different environments. We identified one such environment to be the browser and introduce DuckDB-Wasm, a comprehensive data analysis library for the Web.

Figure 40 presents a difficult trade-off that motivates a more efficient analytical processing in the browser. Web-based analysis tools can either process data locally or on more powerful remote servers. Browsers are limited by the efficiency of the language JavaScript but increase the interactivity by saving costly round-trips over the internet. This contrast asks for a continuous assessment, if the higher efficiency of remote servers justifies higher base latencies. The decision is non-trivial and offers the popular escape-hatch to always run the entire analysis remotely. DuckDB-Wasm accelerates the data processing in browsers and sheds new light on local processing as driver for interactive analytics.

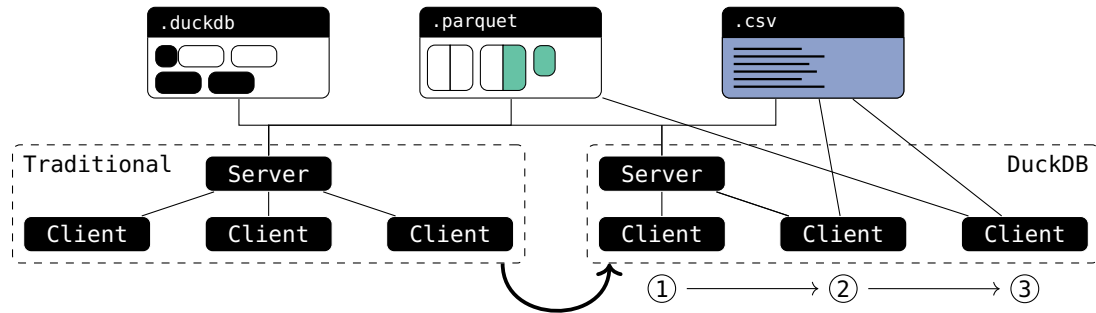
This chapter demonstrates the concept of WebAssembly-driven analytics. We give an overview about the design of DuckDB-Wasm in Section 6.2. Section 6.3 compares the performance with existing libraries in the Web. Section 6.4 demonstrates the capabilities of DuckDB-Wasm using an interactive SQL shell in the browser. We close with a summary of the chapter in Section 6.5.

6.2 DESIGN AND IMPLEMENTATION

In this section, we introduce four key design aspects of DuckDB-Wasm. We describe the interaction with WebAssembly, a browser-agnostic web filesystem, the role of web workers and the efficient integration of user-defined functions.

6.2.1 Embedding WebAssembly

We translate DuckDB to WebAssembly using the compiler *Emscripten* that builds on the LLVM framework [137]. DuckDB is written in C++, a language



```
CREATE VIEW recentdaily AS (
  SELECT date_trunc('day', ts) AS day,
         name, max(price) AS price
  FROM 's3://bucket/stocks.parquet'
  WHERE ts > current_date - 30
  GROUP BY day);
```

```
SELECT r.day, p.name, max(p.count*r.price)
FROM 'https://remote/portfolio.csv' p,
     recentdaily r
WHERE p.name = r.name GROUP BY ALL;
```

Figure 41: A SQL script that downloads stock data from AWS S3 stored in a Parquet file and joins it with a portfolio stored in a CSV file. The left side presents multiple ways to execute the script in a distributed setting. ① shows the traditional separation between client and server, ③ a fully local execution, ② a hybrid mode in between.

that differs significantly from JavaScript in areas such as function calls, data types and memory ownership. WebAssembly does not conceal these language differences but pronounces them further through the memory isolation towards the JavaScript heap. DuckDB-Wasm therefore uses Arrow for efficient data exchange between the two languages. Arrow is a columnar format that is organized in chunks of column vectors, called record batches, and supports zero-copy reads with a small overhead. DuckDB-Wasm serializes results as Arrow IPC streams in C++ and then reads them directly from the WebAssembly heap using JavaScript.

6.2.2 Web Filesystem

DuckDB-Wasm integrates a filesystem that is agnostic to the browser environment. DuckDB is built on top of a virtual filesystem that decouples higher level tasks, such as reading a Parquet file, from low-level filesystem APIs that are specific to the operating system. We use this abstraction in DuckDB-Wasm to tailor filesystem implementations to the different WebAssembly environments, such as the browser and Node.js. In the browser, file opera-

tions are mapped either to the web File API for local files or synchronous XMLHttpRequests for remote data. We use the HTTP range header to request parts of remote files and maintain exponentially growing readahead buffers to reduce the total number of requests.

Figure 41 shows an example script with two SQL statements. The script filters stock price data of the last 30 days, stored in a Parquet file on AWS S3. Afterwards, it joins the data with a stock portfolio in a CSV file that is specified as raw HTTP URL. The **green** color hints at relevant stock prices in the Parquet file and indicates, that DuckDB-Wasm can skip row groups based on the filter predicate. The CSV file is colored in **blue** and is fetched completely.

The figure also displays different execution strategies for the two statements. Traditionally, the capabilities of browsers have been limited, favoring server-based analytics. With this model, a rising number of clients increases the load on the infrastructure and demands for elastic scaling. DuckDB-Wasm, in contrast, offers a choice between the three options labeled with ① to ③. ① adopts the traditional approach where the entire computation would be done by dedicated servers. ③ presents distributed computations where every client runs the analysis locally. ② combines both approaches by aggregating and filtering stock data using a server and joining the result with the portfolio on the local device.

6.2.3 Web Workers

The format Arrow also facilitates the offloading of DuckDB-Wasm to dedicated web workers as we can pass Arrow buffers efficiently through the browser's message API. We use web workers for multiple reasons. First, they unblock the browser's main event loop and allow running complex analytical queries without pausing user interface updates. Second, DuckDB-Wasm can select between worker versions dynamically. Since the release of WebAssembly back in 2017, which is now referred to as MVP, the standard has been evolving. New features, such as WebAssembly Exceptions and SIMD, find their way into the browsers at different speeds, creating a fractured space of post-MVP functionality. These features can bring flat performance improvements and are indispensable when aiming for a maximum speed. We compile DuckDB-Wasm with multiple feature profiles and select a worker based on dynamic browser checks.

6.2.4 User-Defined Functions

DuckDB-Wasm further simplifies the interaction with JavaScript through user-defined functions. DuckDB follows a vectorized execution model and processes queries chunk-wise to amortize the overhead of query interpretation and benefit from superscalar capabilities of the CPU. We use this vectorization to implement efficient user-defined functions in the browser. Users can register JavaScript functions in DuckDB-Wasm and reference them within a SQL query. During execution, the runtime system reads the current chunk data directly from the WebAssembly heap and passes the tuples to the user function in a compact loop.

6.3 TPC-H BENCHMARK

In this section, we experimentally evaluate analytical query processing with DuckDB-Wasm using the TPC-H benchmark. Our experiments were performed on a Ryzen 5800X CPU with Node.js v17.6.0 that is powered by the V8 engine v9.6.

We compare the execution times of TPC-H queries using DuckDB-Wasm and the systems SQL.js, Arquero, and Lovefield. SQL.js is the WebAssembly version of the database SQLite and supports all TPC-H queries out of the box. Lovefield only supports a custom SQL-like API but optimizes query plans internally. However, Lovefield does not support arithmetic operations and nested subqueries within the plan which makes it difficult to run more complex TPC-H queries. Arquero only provides a DataFrame-like API without any upfront optimization. We therefore constructed the TPC-H queries manually for Arquero using the optimized plans produced by the optimizer of a relational database.

We ran the benchmark at the scale factors 0.01, 0.1, and 0.5. A scale factor of 0.1 refers to approximately 100 MB of combined data, resulting in a range between 10 to 500 MB in the experiment. The WebAssembly memory is currently capped at 4 GB in browsers, leaving some room for higher scale factors. We omitted them in the benchmark because of the already significant differences between the systems at scale factor 0.5. Table 7 lists the execution times of the first 14 TPC-H queries. The table also shows the geometric means using a subset of all 22 queries, that are supported by every system. DuckDB-Wasm outperforms the competition by a factor of 10 to 100 across all scale factors.

Table 7: Execution times in seconds for TPC-H queries at the scale factors 0.01, 0.1 and 0.5.

#	SF = 0.01					SF = 0.1					SF = 0.5				
	DuckDB	SQL.js	Arquero	Lovefield	Lovefield	DuckDB	SQL.js	Arquero	Lovefield	Lovefield	DuckDB	SQL.js	Arquero	Arquero	Lovefield
1	0.005	0.054	0.063	0.046	0.046	0.047	0.584	0.823	0.805	0.805	0.235	3.412	9.080	9.080	4.979
2	0.002	0.002	0.003	—	—	0.005	0.019	0.122	—	—	0.015	0.101	3.314	3.314	—
3	0.002	0.014	0.047	0.020	0.020	0.008	0.150	0.570	0.281	0.281	0.048	0.791	5.923	5.923	1.626
4	0.001	0.003	0.028	0.014	0.014	0.008	0.033	0.361	0.234	0.234	0.048	0.181	2.060	2.060	1.573
5	0.003	0.013	0.020	0.008	0.008	0.008	0.153	0.539	0.178	0.178	0.049	0.875	9.498	9.498	1.415
6	0.001	0.010	0.009	0.007	0.007	0.005	0.100	0.107	0.121	0.121	0.026	0.532	0.622	0.622	0.793
7	0.003	0.017	0.049	0.016	0.016	0.016	0.202	0.491	0.498	0.498	0.086	1.174	2.574	2.574	12.391
8	0.004	0.020	0.016	0.008	0.008	0.010	0.288	0.157	0.189	0.189	0.070	1.831	0.894	0.894	1.399
9	0.006	0.027	0.716	0.211	0.211	0.057	0.481	—	3.243	3.243	0.483	3.317	—	—	—
10	0.003	0.010	0.029	0.013	0.013	0.020	0.106	0.420	0.270	0.270	0.116	0.559	7.784	7.784	1.678
11	0.001	0.004	0.001	—	—	0.003	0.050	0.007	—	—	0.007	0.265	0.039	0.039	—
12	0.003	0.009	0.012	0.017	0.017	0.019	0.096	0.154	0.277	0.277	0.089	0.493	1.992	1.992	1.704
13	0.002	0.020	0.012	0.044	0.044	0.014	0.327	0.197	0.572	0.572	0.068	2.246	5.200	5.200	3.251
14	0.001	0.009	0.039	0.013	0.013	0.005	0.094	0.405	0.223	0.223	0.024	0.498	2.261	2.261	1.475
\varnothing_{geo}	0.003	0.012	0.023	0.019	0.019	0.013	0.142	0.268	0.338	0.338	0.073	0.809	2.822	2.822	2.049



Figure 42: A shell that runs entirely in the browser and evaluates SQL queries using DuckDB-Wasm. The figure shows a query joining two parquet files with the relations *orders* and *customer* of the TPC-H benchmark at scale factor 0.1. ① lists the query results and page accesses, ② shows the query plan.

The two JavaScript libraries Arquero and Lovefield scale worse with a growing amount of data compared to the two WebAssembly systems.

The experiment confirms, that WebAssembly enables efficient data processing in the browser. It also shows that DuckDB-Wasm offers sub-second execution times for complex analytical queries on data sizes that may be considered large for the Web. We want to emphasize that DuckDB-Wasm does not substitute existing database systems when processing large amounts of data. Instead, DuckDB-Wasm aims to complement database servers to increase the interactivity for browser-manageable data subsets.

6.4 DEMONSTRATION SCENARIO

We demonstrate the capabilities of a WebAssembly database with an interactive SQL shell that runs entirely in the user’s browser. The SQL shell is accessible at ankoh.github.io/duckdb-wasm and provides a command prompt for a local DuckDB-Wasm instance.

We invite the reader to explore the remote TPC-H data and their own local files ad-hoc in the browser using arbitrary SQL queries. We further propose to reproduce the following three observations: First, when scanning a Parquet file with a limit clause, DuckDB-Wasm only reads the metadata in the back of the file and the first bytes of required columns in the front. Second, aggregates like the global tuple count can be evaluated entirely on the Parquet metadata and finish quickly even on large remote files. Third, when fully scanning a

file, DuckDB-Wasm reads ranges of exponentially increasing sizes to reduce the overhead of individual reads.

Figure 42 shows the WebAssembly shell in action. The figure presents the execution of a query that joins data from two local Parquet files. The files store the relations *orders* and *customers* from the TPC-H benchmark at scale factor 0.1. *orders.parquet* contains 150'000 tuples and measures 11.8 MB. *customer.parquet* contains 15'000 tuples and accounts for 2.6 MB. ② shows the query plan that consists of two Parquet scans, a hash join on the customer key and a topmost projection. DuckDB-Wasm executes the query in 40 milliseconds with cold caches and in 6 milliseconds afterwards. ① also prints paging information of the customer data after running the query twice. It shows that DuckDB-Wasm reads 475 KB in total for the metadata in the back of the file and the required attributes in the front.

6.5 SUMMARY

In this chapter, we introduced DuckDB-Wasm, a WebAssembly version of the database system DuckDB that provides fast analytical processing for the web. We outlined implementation details and showed that DuckDB-Wasm outperforms existing systems by a large margin in the TPC-H benchmark. The demonstration scenario presents an interactive shell that allows executing analytical SQL statements in the local browser. Nevertheless, we identify two major opportunities for future improvements.

First, we believe that WebAssembly unveils hitherto dormant potential for shared query processing between clients and servers. Pushing computation closer to the client eliminates costly round-trips over the internet and thus increases interactivity and scalability of in-browser analytics. However, client-sided analytics also stresses the importance of data locality and asks for a thorough optimization of distributed query plans. Distributed query plans should take into account where data is located, how computation and bandwidth resources can be scaled, and how query latencies evolve during interactive and repeated executions. We see the tandem of DuckDB and DuckDB-Wasm as a first step towards a universal data plane spanning across traditional database servers, clients, CDN workers, and computational storage.

Second, DuckDB-Wasm barely scratches the surface of efficient browser-agnostic data processing. The browser landscape is evolving with new APIs

and WebAssembly capabilities at the horizon. Extensive filesystem support, for example, will evolve DuckDB-Wasm from an in-memory analytical query engine to a persistent database system that can bypass browser memory limitations completely through out-of-core operators. WebAssembly Module Linking will further facilitate the dynamic loading of DuckDB extensions for ICU timezones and full-text search. Additionally, multithreading in browsers has been hampered by the repercussions of the Spectre and Meltdown vulnerabilities [61, 77]. DuckDB scales seamlessly to a large number of cores outside of WebAssembly which could accelerate in-browser analytics even further in the future.

7 | CONCLUSION

This thesis demonstrates the capability of relational database systems to execute complete analysis workflows. We proposed a language extension for analysis workflows and examined three options to improve the functionality and latency of existing systems. This chapter reviews the contributions, describes their limitations and outlines future research directions.

7.1 REVIEW OF THESIS CONTRIBUTIONS

We introduce the language DashQL in Chapter 3 as an extension to SQL that facilitates optimizations between previously loosely connected domains. The coherent language model of DashQL extends known techniques, such as projection and predicate pushdown, in two directions, towards the visualization of query results and the preparation of remote data. AM₄ is an example of a visualization-driven aggregation that combines knowledge about the data model and the embedding workflow context to accelerate the visualization of time series data significantly. The thesis optimizes analytical workflows in Chapter 4 by modularizing aggregation operators within relational database systems. Low-level plan operators decompose traditional aggregation operators into DAG-shaped computation graphs and simplify the reasoning about complex and composed aggregation functions. For interactive workflows, we show in Chapter 5 that adaptive query execution significantly reduces the latency of compilation based query engines. These engines can pair the high processing efficiency of an optimized query program with the low latency of unoptimized variants to launch the query execution quickly and update to more efficient versions whenever necessary. Finally, the thesis eliminates round-trips to central analytics servers in Chapter 6 by pushing computation closer to the user with WebAssembly. DuckDB-Wasm is an embedded analytical database for the web that outperforms existing data processing libraries in that space by orders of magnitude.

7.2 LIMITATIONS OF THE SYSTEMS

Nevertheless, the language DashQL is limited in flexibility. We propose a grammar in Chapter 3 that provides several extension points for declaring new input types, importing new data sources, loading different formats or abbreviating visualizations. However, DashQL remains a SQL dialect for declarative analysis workflows and is more constrained than an imperative scripting language. DashQL workflows enable holistic optimizations and seamless mappings to computation graphs but do not offer rich control-flow constructs for branches and loops. They further cannot make use of arbitrary external libraries without mapping them to the task graph first. We hope to embed external libraries more quickly in the future by controlling the effects that they may have on existing data.

Additionally, reducing latency and maximizing throughput appear as contrary goals to adaptive query execution. In Chapter 5, we selected multiple execution modes based on the number of assigned threads, the expected compilation times, and the expected speedups. Based on these estimates, the system would often start the execution with a less efficient mode and later upgrade pipelines to higher efficiency, if necessary. When using LLVM, upgrading is done on a single thread which means that a fast start involves other threads more quickly. This approach optimizes for the shortest execution times of individual queries at the expense of less efficient use of the hardware during the upgrade. When executing multiple queries simultaneously, the system should not waste precious CPU cycles on less efficient execution modes but instead optimize more pipelines upfront. Our cost model hides this contrast through the number of assigned threads, which will require careful and non-trivial coordination with the query scheduler.

7.3 FUTURE DIRECTIONS

The thesis contributes to a tighter integration of relational database management systems into analytical workflows. We see the following three directions for future research that build upon the introduced concepts.

7.3.1 Distributing Workflows

DashQL workflows can be distributed. We describe DashQL in Chapter 3 as a language that evaluates analytical workflows interactively by maintaining a task graph and updating the workflow state based on user interactions. We outline optimizations that increase workflow efficiency by propagating valuable information between the statements. Similarly, the coherent language model would enable informed decisions about the distribution of a workflow based on data sizes, data proximity, and query complexity. This bridges the gap between traditional research around distributed query execution and workflow system architectures that spread different parts of a workflow across machines.

7.3.2 Advanced Aggregation

We introduced low-Level plan operators in Chapter 4 that modularize relational algebra operators for advanced and composed aggregation functions. The chapter describes a canonical translation of traditional query plans based on heuristic optimization rules. In contrast, text-book optimizations of relational algebra often use cost models and derive optimal plans using dynamic programming. These cost models are not directly applicable to low-level plan operators since they do not account for the added physical properties of intermediate results. It would be interesting to construct a cost model that includes partitioning and ordering costs to improve the plan quality.

Additionally, the chapter focuses on the efficient evaluation and composition of associative aggregates and window functions. Low-level plan operators can also be used to implement even more complex aggregation functions such as `MATCH_RECOGNIZE` introduced in SQL:2016. `MATCH_RECOGNIZE` resembles SQL windows in that aggregates are computed per row and respect explicit partitioning and ordering clauses. However, it also adds filtering of values based on patterns defined on SQL expressions that may require optimizations found in regular expression engines. `MATCH_RECOGNIZE` therefore serves as a good example where low-level plan operators may already implement and optimize parts of the required functionality and only need a specific operator to implement the matching.

7.3.3 Latency-driven Optimization

Database systems should optimize for end-to-end latencies. Chapter 5 mentioned the PostgreSQL administration tool *pgAdmin* that runs dozens of queries at startup, joining only small metadata tables. In such scenarios, optimizing for the highest processing efficiency can have detrimental effects if higher base latencies dominate the execution time of cheap queries.

We see this as an example of misaligned optimization goals in the larger context of analytical applications: Databases have traditionally lacked information about the application context and have therefore optimized queries defensively and in isolation. This has led to a suboptimal performance which is underpinned by the widespread use of external data caching in front of databases for fast result reuse and refinement. The effects are severe as workarounds, such as external caching, drag applications into data management territory and expose them to consistency requirements better handled within a database system. This motivates the question of whether today's database systems should not optimize queries concerning the surrounding application context. Doing so would shift the focus from optimizing individual queries in isolation to minimizing end-to-end latencies across multiple queries.

7.4 CONCLUDING REMARKS

Our ever-expanding digital footprints generate an increasing amount of data that ought to be summarized, explored, shared, and explained. Relational database systems form the backbone of these efforts as they are indispensable for efficiently persisting and querying this data. In the past, however, database systems have operated in isolation without including the broader context of data analysis workflows. We broke the isolation in this thesis through extended language support, reduced latencies and advanced analytical functions. I see the interoperability with external tools as a distinguishing functionality of tomorrow's database services and expect the overlap between the research fields to grow in the future.

BIBLIOGRAPHY

- [1] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. "Database Languages for Sensor Networks". In: *Encyclopedia of Database Systems* 3 (2018), pp. 951–956.
- [2] Sameer Agarwal, Davies Liu, and Reynold Xin. *Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop*. "https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html". 2016.
- [3] Josep Aguilar-Saborit, Raghu Ramakrishnan, Krish Srinivasan, Kevin Bocksrocker, Ioannis Alagiannis, Mahadevan Sankara, Moe Shafiei, Jose Blakeley, Girish Dasarathy, Sumeet Dash, et al. "POLARIS: the distributed SQL engine in azure synapse". In: *PVLDB* 12 (2020), pp. 3204–3216.
- [4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. "Massively parallel sort-merge joins in main memory multi-core database systems". In: *PVLDB* 10 (2012), pp. 1064–1075.
- [5] C. Balkesen, J. Teubner, G. Alonso, and M. T. Ozsü. "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware". In: *ICDE*. 2013.
- [6] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsü. "Multi-core, main-memory joins". In: *PVLDB* 1 (2013), pp. 85–96.
- [7] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer ozsü. "Main-Memory Hash Joins on Modern Processor Architectures". In: *TKDE* 7 (2015), pp. 1754–1766.
- [8] Maximilian Bandle, Jana Giceva, and Thomas Neumann. "To Partition, or Not to Partition, That is the Join Question in a Real System". In: *SIGMOD*. 2021.
- [9] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. "Memory-efficient Hash Joins". In: *PVLDB* 4 (2014), pp. 353–364.

- [10] Leilani Battle, Remco Chang, and Michael Stonebraker. "Dynamic Prefetching of Data Tiles for Interactive Visualization". In: *SIGMOD*. 2016, pp. 1363–1375.
- [11] Leilani Battle and Carlos Scheidegger. "A Structured Review of Data Management Technology for Interactive Visualization and Analysis". In: *IEEE Transactions on Visualization and Computer Graphics* 2 (2021), pp. 1128–1138.
- [12] Veronique Benzaken, Jean-Daniel Fekete, Pierre-Luc Hemery, Wael Khemiri, and Ioana Manolescu. "EdiFlow: Data-intensive interactive workflows for visual analytics". In: *ICDE*. 2011.
- [13] Jacques Bertin. *Semiology of Graphics*. 1983.
- [14] Spyros Blanas, Yinan Li, and Jignesh M. Patel. "Design and evaluation of main memory hash join algorithms for multi-core CPUs". In: *SIGMOD*. 2011.
- [15] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution". In: *CIDR*. 2005.
- [16] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. "D3 Data-Driven Documents". In: *IEEE Transactions on Visualization and Computer Graphics* 12 (2011), pp. 2301–2309.
- [17] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. "Generating custom code for efficient query execution on heterogeneous processors". In: *The VLDB Journal* 6 (2018), pp. 797–822.
- [18] Jesús Camacho-Rodríguez et al. "Apache Hive: From MapReduce to Enterprise-Grade Big Data Warehousing". In: *SIGMOD*. 2019.
- [19] Yu Cao, Chee-Yong Chan, Jie Li, and Kian-Lee Tan. "Optimization of analytic window functions". In: *PVLDB* 11 (2012), pp. 1244–1255.
- [20] Donald D. Chamberlin and Raymond F. Boyce. "SEQUEL: A structured English query language". In: *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*. 1976, pp. 249–264.
- [21] Craig Chasseur and Jignesh M. Patel. "Design and evaluation of storage organizations for read-optimized main memory databases". In: *PVLDB* 13 (2013), pp. 1474–1485.

- [22] Biswapesh Chattopadhyay et al. “Procella: Unifying serving and analytical data at Youtube”. In: *PVLDB* 12 (2019), pp. 2022–2034.
- [23] Edward Y Chen, Christopher M Tan, Yan Kou, Qiaonan Duan, Zichen Wang, Gabriela Vaz Meirelles, Neil R Clark, and Avi Ma’ayan. “Enrichr: interactive and collaborative HTML5 gene list enrichment analysis tool”. In: *BMC bioinformatics* 1 (2013), pp. 1–14.
- [24] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. “Efficient implementation of sorting on multi-core SIMD CPU architecture”. In: *PVLDB* 2 (2008), pp. 1313–1324.
- [25] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Software Pioneers* 6 (2002), pp. 263–294.
- [26] Edgar F Codd. “A relational model of data for large shared data banks”. In: *Software pioneers*. 2002, pp. 263–294.
- [27] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. “An architecture for compiling UDF-centric workflows”. In: *PVLDB* 12 (2015), pp. 1466–1477.
- [28] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. “Tuplware: “Big” Data, Big Analytics, Small Clusters”. In: *CIDR*. 2015.
- [29] Benoit Dageville et al. “The Snowflake Elastic Data Warehouse”. In: *SIGMOD*. 2016.
- [30] Benjamin De Boe, Tom Woodfin, Thomas Dyar, Dave McCaldon, Aleks Djakovic, Alex MacLeod, and Don Woodlock. “IntegratedML: Every SQL Developer is a Data Scientist”. In: *DEEM’20*. 2020.
- [31] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. “Hekaton: SQL server’s memory-optimized OLTP engine”. In: *SIGMOD*. 2013.
- [32] Jens Dittrich and Joris Nix. “The Case for Deep Query Optimisation”. In: *CIDR*. 2020.
- [33] Stefan Edelkamp and Armin Weiß. “BlockQuicksort: Avoid Branch Mispredictions in Quicksort”. In: *ACM Journal of Experimental Algorithmics* 1 (2019), pp. 1–22.
- [34] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. “Spinning fast iterative data flows”. In: *PVLDB* 11 (2012), pp. 1268–1279.

- [35] Ziqiang Feng and Eric Lo. “Accelerating aggregation using intra-cycle parallelism”. In: *ICDE*. 2015.
- [36] Wojciech Fraczak, Loukas Georgiadis, Andrew Miller, and Robert E. Tarjan. “Finding dominators via disjoint set union”. In: *Journal of Discrete Algorithms* (2013), pp. 2–20.
- [37] Craig Freedman, Erik Ismert, and Per-Åke Larson. “Dynamic SQL”. In: *IEEE Data Eng. Bull.* 1 (2019).
- [38] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. “Adopting worst-case optimal joins in relational database systems”. In: *PVLDB* 12 (2020), pp. 1891–1904.
- [39] Johann Christoph Freytag. “A Rule-Based View of Query Optimization”. In: *SIGMOD*. 3. 1987, pp. 173–180.
- [40] Sunny Gakhar et al. “Pipemizer: An Optimizer for Analytics Data Pipelines”. In: *PVLDB*. 2022, pp. 95–109.
- [41] Loukas Georgiadis, Robert E. Tarjan, and Renato F. Werneck. “Finding Dominators in Practice”. In: *Journal of Graph Algorithms and Applications* 1 (2006), pp. 69–94.
- [42] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and David Sculley. “Google vizier: A service for black-box optimization”. In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 2017, pp. 1487–1495.
- [43] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. “Maintenance of cube automatic summary tables”. In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data - SIGMOD '00* 1 (2000).
- [44] Torsten Grust. “Accelerating XPath location steps”. In: *SIGMOD*. 2002.
- [45] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. “Amazon Redshift and the Case for Simpler Data Warehouses”. In: *SIGMOD*. 2015.
- [46] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. “Bringing the web up to speed with WebAssembly”. In: *ACM SIGPLAN Notices* 6 (2017), pp. 185–200.

- [47] Sibsankar Haldar. *Inside sqlite*. 2007.
- [48] Jeffrey Heer, Joseph M Hellerstein, and Sean Kandel. “Predictive Interaction for Data Transformation.” In: *CIDR*. 2015.
- [49] Joseph M. Hellerstein et al. “The MADlib analytics library”. In: *PVLDB* 12 (2012), pp. 1700–1711.
- [50] Brian Hempel, Justin Lubin, and Ravi Chugh. “Sketch-n-Sketch: Output-Directed Programming for SVG”. In: *UIST*. 2019, pp. 281–292.
- [51] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. “Visual Debugging Techniques for Reactive Data Visualization”. In: *Computer Graphics Forum*. 3. 2016, pp. 271–280.
- [52] Shrainik Jain, Edward D. Lazowska, Bill Howe, Dan C. Halperin, and Dominik Moritz. “SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment”. In: *SIGMOD*. 2016.
- [53] Abdallah Jarwan, Ayman Sabbah, and Mohamed Ibnkahla. “Information-Oriented Traffic Management for Energy-Efficient and Loss-Resilient IoT Systems”. In: *IEEE Internet of Things Journal* 10 (2022), pp. 7388–7403.
- [54] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. “M4: A Visualization-Oriented Time Series Data Aggregation”. In: *PVLDB* 10 (2014), pp. 797–808.
- [55] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. “Enterprise Data Analysis and Visualization: An Interview Study”. In: *IEEE Transactions on Visualization and Computer Graphics* 12 (2012), pp. 2917–2926.
- [56] Manos Karpathiotakis, Ioannis Alagiannis, Thomas Heinis, Miguel Branco, and Anastasia Ailamaki. “Just-In-Time Data Virtualization: Lightweight Data Management with ViDa”. In: *CIDR*. 2015.
- [57] Timo Kersten, Viktor Leis, and Thomas Neumann. “Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra”. In: *The VLDB Journal* 5 (2021), pp. 883–905.
- [58] Mary Beth Kery, Donghao Ren, Fred Hohman, Dominik Moritz, Kanit Wongsuphasawat, and Kayur Patel. “mage: Fluid Moves Between Code and Graphical Work in Computational Notebooks”. In: *UIST*. 2020, pp. 140–151.

- [59] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. “Building efficient query engines in a high-level language”. In: *PVLDB* 10 (2014), pp. 853–864.
- [60] Christoph Koch, Yanif Ahmad, Oliver Kennedy, Milos Nikolic, Andres Nötzli, Daniel Lupei, and Amir Shaikhha. “DBToaster: higher-order delta processing for dynamic, frequently fresh views”. In: *The VLDB Journal* 2 (2014), pp. 253–278.
- [61] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *S&P*. 2019.
- [62] André Kohn. “Adaptive Execution of Compiled Queries”. MA thesis. Technical University of Munich, 2017.
- [63] Andre Kohn, Viktor Leis, and Thomas Neumann. “Making Compiling Query Engines Practical”. In: *TKDE* 1 (2019), pp. 1–1.
- [64] André Kohn, Viktor Leis, and Thomas Neumann. “Adaptive Execution of Compiled Queries”. In: *ICDE*. Apr. 2018, pp. 197–208.
- [65] André Kohn, Viktor Leis, and Thomas Neumann. “Building Advanced SQL Analytics From Low-Level Plan Operators”. In: *SIGMOD*. 2021, pp. 1001–1013.
- [66] André Kohn, Viktor Leis, and Thomas Neumann. “Making Compiling Query Engines Practical”. In: *TKDE* 2 (2021), pp. 597–612.
- [67] André Kohn, Dominik Moritz, Mark Raasveldt, Hannes Mühleisen, and Thomas Neumann. “DuckDB-Wasm: Fast Analytical Processing for the Web”. In: *PVLDB*. 12. 2022, pp. 3574–3577.
- [68] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. “Generating code for holistic query evaluation”. In: *ICDE*. 2010.
- [69] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. “Massively Parallel NUMA-Aware Hash Joins”. In: *IMDM*. 2015, pp. 3–14.
- [70] Per-Åke Larson, Cipri Clinciu, Eric N. Hanson, Artem Oks, Susan L. Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. “SQL server column store indexes”. In: *SIGMOD*. 2011.
- [71] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. “Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age”. In: *SIGMOD*. 2014.

- [72] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. "How good are query optimizers, really?" In: *PVLDB* 3 (2015), pp. 204–215.
- [73] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. "Efficient processing of window functions in analytical SQL queries". In: *PVLDB* 10 (2015), pp. 1058–1069.
- [74] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. "OLTP through the looking glass, and what we found there". In: *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker* (2018), pp. 409–439.
- [75] Alexander Lex, Nils Gehlenborg, Hendrik Strobelt, Romain Vuillemot, and Hanspeter Pfister. "UpSet: visualization of intersecting sets". In: *IEEE transactions on visualization and computer graphics* 12 (2014), pp. 1983–1992.
- [76] Lauro Lins, James T. Klosowski, and Carlos Scheidegger. "Nanocubes for Real-Time Exploration of Spatiotemporal Datasets". In: *IEEE Transactions on Visualization and Computer Graphics* 12 (2013), pp. 2456–2465.
- [77] Moritz Lipp et al. "Meltdown". In: *USENIX*. 6. 2020, pp. 46–56.
- [78] Zhicheng Liu and Jeffrey Heer. "The Effects of Interactive Latency on Exploratory Visual Analysis". In: *IEEE Transactions on Visualization and Computer Graphics* 12 (2014), pp. 2122–2131.
- [79] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. "imMens: Real-time Visual Querying of Big Data". In: *Computer Graphics Forum* 3pt4 (2013), pp. 421–430.
- [80] Guy M. Lohman. "Grammar-like functional rules for representing query optimization alternatives". In: *SIGMOD*. 3. 1988, pp. 18–27.
- [81] Stefan Manegold, Peter Boncz, and Martin Kersten. "Optimizing database architecture for the new bottleneck: Memory access". In: *The VLDB Journal* (Dec. 2000), pp. 231–246.
- [82] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. "Relaxed operator fusion for in-memory databases". In: *PVLDB* 1 (2017), pp. 1–13.
- [83] Dominik Moritz, Jeffrey Heer, and Bill Howe. "Dynamic Client-Server Optimization for Scalable Interactive Visualization on the Web". In: *DSIA*. 2015.

- [84] Dominik Moritz, Bill Howe, and Jeffrey Heer. “Falcon: Balancing Interactive Latency and Resolution Sensitivity for Scalable Linked Visualizations”. In: *CHI*. 2019.
- [85] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. “Cache-Efficient Aggregation: Hashing Is Sorting”. In: *SIGMOD*. 2015.
- [86] DAVID R. MUSSER. “Introspective Sorting and Selection Algorithms”. In: *Software: Practice and Experience* 8 (1997), pp. 983–993.
- [87] Thomas Neumann. “Efficiently compiling efficient query plans for modern hardware”. In: *PVLDB* 9 (2011), pp. 539–550.
- [88] Thomas Neumann and Michael J. Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *CIDR*. 2020.
- [89] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Malte Schwarzkopf, Saman P. Amarasinghe, and Matei Zaharia. “Weld: A Common Runtime for High Performance Data Analysis”. In: *CIDR*. 2017.
- [90] Drew Paroski. *Code Generation: The Inner Sanctum Of Database Performance*. "<http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-of-database-performance.html>". 2016.
- [91] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. “Quickstep: A Data Platform Based on the Scaling-Up Approach”. In: *PVLDB* 6 (2018), pp. 663–676.
- [92] Andrew Pavlo et al. “Self-Driving Database Management Systems”. In: *CIDR*. 2017.
- [93] Fernando Magno Quintão Pereira and Jens Palsberg. “SSA Elimination after Register Allocation”. In: *Lecture Notes in Computer Science* (2009), pp. 158–173.
- [94] Duy-Hung Phan and Pietro Michiardi. “A novel, low-latency algorithm for multiple Group-By query optimization”. In: *ICDE*. 2016.
- [95] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. “Voodoo - a vector algebra for portable database performance on modern hardware”. In: *PVLDB* 14 (2016), pp. 1707–1718.
- [96] Massimiliano Poletto and Vivek Sarkar. “Linear scan register allocation”. In: *ACM Transactions on Programming Languages and Systems* 5 (1999), pp. 895–913.

- [97] Orestis Polychroniou and Kenneth A. Ross. “High throughput heavy hitter aggregation for modern SIMD processors”. In: *DaMoN*. 2013.
- [98] Fotis Psallidas and Eugene Wu. “Provenance for Interactive Visualizations”. In: *HILDA*. 2018.
- [99] Mark Raasveldt and Hannes Mühleisen. “Data Management for Data Science - Towards Embedded Analytics”. In: *CIDR*. 2020.
- [100] Mark Raasveldt and Hannes Mühleisen. “DuckDB: An Embeddable Analytical Database”. In: *SIGMOD*. 2019.
- [101] Vijayshankar Raman et al. “DB2 with BLU acceleration”. In: *PVLDB 11* (2013), pp. 1080–1091.
- [102] Bart Samwel et al. “F1 Query: Declarative Querying at Scale”. In: *PVLDB 12* (2018), pp. 1835–1848.
- [103] Arvind Satyanarayan and Jeffrey Heer. “Lyra: An Interactive Visualization Design Environment”. In: *Computer Graphics Forum*. 3. 2014, pp. 351–360.
- [104] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. “Vega-Lite: A Grammar of Interactive Graphics”. In: *IEEE transactions on visualization and computer graphics* (2016), pp. 341–350.
- [105] Arvind Satyanarayan, Ryan Russell, Jane Hoffswell, and Jeffrey Heer. “Reactive Vega: A Streaming Dataflow Architecture for Declarative Interactive Visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 1 (2016), pp. 659–668.
- [106] Stefan Schuh, Xiao Chen, and Jens Dittrich. “An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory”. In: *SIGMOD*. 2016.
- [107] Maximilian Schüle, Tobias Götz, Alfons Kemper, and Thomas Neumann. “ArrayQL for Linear Algebra within Umbra”. In: *SSDBM*. 2021.
- [108] Robert Sedgewick. “Implementing Quicksort programs”. In: *Communications of the ACM* 10 (1978), pp. 847–857.
- [109] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. “Access path selection in a relational database management system”. In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. Boston, Massachusetts, 1979, pp. 511–522.

- [110] Raghav Sethi et al. “Presto: SQL on Everything”. In: *ICDE*. 2019.
- [111] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. “How to Architect a Query Compiler”. In: *SIGMOD*. 2016.
- [112] Jie Shen, Ana Lucia Varbanescu, Henk Sips, Michael Arntzen, and Dick G. Simons. “Glinda: A Framework for Accelerating Imbalanced Applications on Heterogeneous Platforms”. In: *Proceedings of the ACM International Conference on Computing Frontiers*. 2013.
- [113] Carson Sievert and Kenneth Shirley. “LDAvis: A method for visualizing and interpreting topics”. In: *Proceedings of the workshop on interactive language learning, visualization, and interfaces*. 2014, pp. 63–70.
- [114] Laurynas Šikšnys and Torben Bach Pedersen. “SolveDB: Integrating Optimization Problem Solvers Into SQL Databases”. In: *SSDBM*. 2016.
- [115] Utku Sirin, Pinar Tözün, Danica Porobic, and Anastasia Ailamaki. “Micro-architectural Analysis of In-memory OLTP”. In: *SIGMOD*. 2016.
- [116] Chris Stolte, Diane Tang, and Pat Hanrahan. “Polaris”. In: *Communications of the ACM* 11 (2008), pp. 75–84.
- [117] Rebecca Taft et al. “CockroachDB: The Resilient Geo-Distributed SQL Database”. In: *SIGMOD*. 2020.
- [118] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. “How to Architect a Query Compiler, Revisited”. In: *SIGMOD*. 2018, pp. 307–322.
- [119] Jeffrey Tao, Yiru Chen, and Eugene Wu. “Demonstration of PI2: Interactive Visualization Interface Generation for SQL Analysis in Notebook”. In: *Proceedings of the 2022 International Conference on Management of Data* (2022).
- [120] Pawel Terlecki, Fei Xu, Marianne Shaw, Valeri Kim, and Richard Wesley. “On Improving User Response Times in Tableau”. In: *SIGMOD*. 2015.
- [121] Pinar Tözün, Islam Atta, Anastasia Ailamaki, and Andreas Moshovos. “ADDICT: Advanced Instruction Chasing for Transactions”. In: *PVLDB* 14 (2014), pp. 1893–1904.
- [122] Pinar Tözün, Brian Gold, and Anastasia Ailamaki. “OLTP in wonderland: where do cache misses come from in major OLTP components?”. In: *DaMoN*. 2013.
- [123] Edward R. Tufte. *The Visual Display of Quantitative Information*. USA, 1986.

- [124] Sahithi Tummalapalli and Venkata rao Machavarapu. "Managing Mysql Cluster Data Using Cloudera Impala". In: *Procedia Computer Science* 1 (2016), pp. 463–474.
- [125] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. "Get Real: How Benchmarks Fail to Represent the Real World". In: *DBTest*. 2018.
- [126] Richard Wesley and Fei Xu. "Incremental computation of common windowed holistic aggregates". In: *PVLDB* 12 (2016), pp. 1221–1232.
- [127] Hadley Wickham. "A Layered Grammar of Graphics". In: *Journal of Computational and Graphical Statistics* 1 (2010), pp. 3–28.
- [128] Hadley Wickham. "A layered grammar of graphics". In: *Journal of Computational and Graphical Statistics* 1 (2010), pp. 3–28.
- [129] Leland Wilkinson. "The grammar of graphics". In: *Handbook of computational statistics*. 2012, pp. 375–414.
- [130] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. "Towards a general-purpose query language for visualization recommendation". In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. 2016, pp. 1–6.
- [131] Kanit Wongsuphasawat, Dominik Moritz, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. "Voyager: Exploratory Analysis via Faceted Browsing of Visualization Recommendations". In: *IEEE Transactions on Visualization and Computer Graphics* 1 (2016), pp. 649–658.
- [132] Kanit Wongsuphasawat, Zening Qu, Dominik Moritz, Riley Chang, Felix Ouk, Anushka Anand, Jock Mackinlay, Bill Howe, and Jeffrey Heer. "Voyager 2: Augmenting visual analysis with partial view specifications". In: *Proceedings of the 2017 chi conference on human factors in computing systems*. 2017, pp. 2648–2659.
- [133] Eugene Wu, Fotis Psallidas, Zhengjie Miao, Haoci Zhang, and Laura Rettig. "Combining Design and Performance in a Data Visualization Management System". In: *CIDR*. 2017.

- [134] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. “B2: Bridging code and interactive visualization in computational notebooks”. In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 2020, pp. 152–165.
- [135] Wenjian Xu, Ziqiang Feng, and Eric Lo. “Fast Multi-Column Sorting in Main-Memory Column-Stores”. In: *SIGMOD*. 2016.
- [136] Junran Yang, Hyekang Kevin Joo, Sai S. Yerramreddy, Siyao Li, Dominik Moritz, and Leilani Battle. “Demonstration of VegaPlus: Optimizing Declarative Visualization Languages”. In: *Proceedings of the 2022 International Conference on Management of Data (2022)*.
- [137] Alon Zakai. “Emscripten: an LLVM-to-JavaScript compiler”. In: *SIGPLAN*. 2011.
- [138] Emanuel Zraggen, Alex Galakatos, Andrew Crotty, Jean-Daniel Fekete, and Tim Kraska. “How Progressive Visualizations Affect Exploratory Analysis”. In: *IEEE Transactions on Visualization and Computer Graphics* 8 (2017), pp. 1977–1987.
- [139] Haoci Zhang, Viraj Raj, Thibault Sellam, and Eugene Wu. “Precision Interfaces for Different Modalities”. In: *SIGMOD*. 2018.
- [140] Rui Zhang, Saumya Debray, and Richard T. Snodgrass. “Micro-specialization: dynamic code specialization of database management systems”. In: *International Symposium on Code Generation and Optimization*. 2012.
- [141] Rui Zhang, Richard T. Snodgrass, and Saumya Debray. “Micro-Specialization in DBMSes”. In: *ICDE*. 2012.