Query Compilation for Modern Data Processing Environments

vorgelegt von
M. Sc.
Philipp Marian Grulich
ORCID: 0000-0001-9497-2895

an der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
- Dr. rer. nat. -

Promotionsausschuss:
Vorsitzender: Prof. Dr. Matthias Böhm, TU Berlin
Gutachter: Prof. Dr. Volker Markl, TU Berlin
Gutachter: Prof. Dr. Carsten Binning, TU Darmstadt
Gutachter: Prof. Dr. Stratos Idreos, Harvard University
Tag der wissenschaftlichen Aussprache: 8. November 2023

Berlin 2023

Acknowledgements

I would like to thank my primary advisors, Volker Markl and Steffen Zeuch, for their great feedback and support, which led to this thesis. I also would like to thank Casten Binnig, Statos Ideros, and Matthias Böhm for being part of my committee and providing valuable feedback.

Over the past years, many people have guided and supported me on this journey. My thanks go to the whole Databases and Information Management Group (DIMA) at TU Berlin. During my master's program, I began as a research assistant at DIMA, mentored by Jonas Traub, Sebastian Breß, and Tilmann Rabl. I am thankful for all the advice and help I received and am grateful for your encouragement to pursue my Ph.D.

During my PhD, I had the opportunity to contribute to NebulaStream. I want to thank all team members for their hard work and time spent improving the system. Ankit Chaudhary, Ariane Ziehn, Eleni Tzirita Zacharatou, Shuhao Zhang, Xenofon Chatziliadis, Dwi Prasetyo Adi Nugroho, Varun Pandey, Nils Schubert, Maroua Taghouti, Julius Hülsmann, Anastasiia Kozar, Adrian Michalke, Aljoscha Lepping, Jonas Traub as well as Dimitrios Giouroukis, Haralampos Gavriilidis, Bonaventura Del Monte, Viktor Rosenfeld, and Steffen Zeuch.

Furthermore, I enjoyed the many discussions I had with colleagues at DIMA. I thank Clemens Lutz, Haralampos Gavriilidis, Gabor Gevay, Viktor Rosenfeld, Bonaventura Del Monte, Martin Kiefer, and Alexander Renz-Wieland for their time to discuss research ideas.

I'm especially grateful to the extraordinary students I've worked with. Juliane Verwiebe took over Scotty, and we published multiple publications in collaboration with Jonas Traub. Victor Bleszka, Hoang Mi, Jan Vincent Szlang, Moritz Ruge, Johannes Maeß contributed to NebulaStream. In particular, I'm proud of working with Adrian Michalke, Aljoscha Lepping, and Nils Schubert, who decided to stay at DIMA to pursue a PhD on their own.

Finally, I thank my family and friends who supported me over the last years. My parents, Sabine Grulich and Harald Wilkending, always supported my passion and motivated me to pursue my goals. I also would like to thank Jan Rohe for encouraging me after painful paper rejections and Alina Günther for your partnership and support that has been a cornerstone of my personal and professional life.

Zusammenfassung

In den letzten Jahren haben sich Datenverarbeitungsanwendungen erheblich verändert. Heutzutage verarbeiten datenintensive Anwendungen ständig wachsende Datenmengen in immer höherer Geschwindigkeit. Gleichzeitig hat die Komplexität und Vielfalt an Anwendungsfällen zugenommen. Diese kombinieren relationalen Operatoren, Stream Processing, Maschinelles Lernen und benutzerdefinierte Funktionen (UDFs). Um diesen Anforderungen gerecht zu werden, wurde eine Vielzahl von Datenverarbeitungssystemen entwickelt. Darüber hinaus hat die Forschungsgemeinschaft erhebliche Anstrengungen unternommen, um die Effizienz und Leistung dieser Systeme zu verbessern. Jüngste Arbeiten zeigen jedoch, dass viele moderne Datenverarbeitungssysteme eine suboptimale Leistung erbringen. Insbesondere haben diese Systeme Techniken zur effizienten Verarbeitung nicht übernommen, z. B. Querykompilierung oder Vektorisierung. Stattdessen konzentrieren sie sich auf horizontale Skalierbarkeit, um die Leistung zu steigern, was jedoch die Komplexität und die Kosten im Betrieb erhöht.

In dieser Dissertation verbessern wir die Architektur aktueller Datenverarbeitungssysteme und beheben deren Leistungseinschränkungen. Zu diesem Zweck nutzen wir Querykompilierung als grundlegendes Konzept für eine effiziente Datenverarbeitung. Im Einzelnen leisten wir die folgenden drei Beiträge. Im ersten Schritt passen wir den Querykompilierungsansatz an die Eigenschaften von Stream Processing Workloads an. Unser neuartiger Querykompilierungansatz überwacht Datencharakteristika, erkennt Änderungen und optimiert Queries adaptiv zur Laufzeit. Darüber hinaus präsentieren wir parallelisierungsfähige Operatoren, die moderne Multi-Core-Hardware vollständig ausnutzen können. Darauf aufbauend ermöglichen wir Querykompilierung für die effiziente Ausführung von polyglotten Abfragen, die UDFs in Sprachen wie Java, Java-Script und Python beinhalten. Unser Ansatz kombiniert relationale Operatoren und UDFs in einer einheitlichen Zwischendarstellung, ermöglicht eine ganzheitliche Optimierung über Operatorgrenzen hinweg und übersetzt polyglotte Abfragen in effiziente Codefragmente. Abschließend präsentieren wir ein neuartiges Framework für die Entwicklung von Query Execution Engines, welches die Vorteile von Querykompilierung nutzt, ohne die Produktivität von Entwicklern zu beeinträchtigen. Unser Framework bietet eine imperatives Programmierinterface und nutzt trace-basierte Just-in-Time-Kompilierung, um effizienten Code zu erzeugen. Darüber hinaus bietet es mehrere Kompilierungs-Backends zur Optimierung von bestimmten Anwendungen, z. B. für OLAP Abfragen, Stream Processing oder UDFs.

Zusammenfassend, präsentiert diese Dissertation mehrere grundlegende Bausteine für den Einsatz von Querykompilierung in modernen Datenverarbeitungssystemen. Hierdurch werden Systemengpässe reduziert, die Hardwareauslastung verbessert und die Leistung in verschiedenen Datenverarbeitungsumgebungen erhöht.

Abstract

Over the last decades, the data processing environment significantly changed. Nowadays, data-centric applications process ever-growing volumes of data with increasing velocity. At the same time, data processing pipelines became more complex and diverse. These workloads exceed traditional analytical queries and involve stream processing, machine learning, and user-defined functions (UDFs). A wide range of general-purpose data processing systems have been proposed to address these requirements. However, recent research has shown that many state-of-the-art data processing systems deliver suboptimal performance. These systems are hardware-oblivious and have not yet adopted techniques for efficient query processing from relational data management systems, e.g., query compilation or vectorization. Instead, they use managed runtimes to hide hardware details and focus on horizontal scalability. This decreases hardware utilization and increases deployment complexity and cost.

In this thesis, we revisit the architecture of current data processing systems and propose building blocks to address their performance limitations. We leverage query compilation as a fundamental concept for efficient data processing and expand its application to a wider set of workloads. In particular, we make the following three contributions: First, we adapt query compilation to the unique characteristics of stream processing workloads. We propose an adaptive query compilation approach that monitors data characteristics of streams, detects changes, and re-optimizes queries at runtime. Furthermore, we introduce parallelization-aware stateful operators that fully utilize modern multi-core processors to handle high-velocity streams. Second, we extend query compilation for the efficient execution of polyglot queries that involve UDFs in high-level programming languages like Java, Java-Script, and Python. Our approach combines relational operators and UDFs in one unified intermediate representation. This enables holistic optimization across operator boundaries and the generation of highly efficient code fragments. Third, we introduce a novel framework for the development of query execution engines to benefit from the advantages of query compilation without sacrificing productivity. Our framework provides an imperative implementation interface and leverages trace-based just-in-time compilation to generate efficient code from data processing queries. Furthermore, it provides multiple compilation backends to optimize the query execution for specific workloads, e.g., short-running queries, stream processing, or UDFs.

In summary, this dissertation provides building blocks for a hardware-conscious architecture of modern data processing systems that mitigate current system bottlenecks, enhance hardware utilization, and improve performance in diverse data processing environments. To this end, we leverage query compilation and adapt it to a wide range of workloads. Finally, our contributions form the basis for new data processing systems like NebulaStream. They improve the query execution performance and reduce the system complexity at the same time.

Table of Contents

1 Introduction

The digital revolution has led to a diverse data management environment enabling new applications, research fields, and business models [137, 173]. Research projects like the Large Hadron Collider [35] and platforms like Google [115] and Facebook [66] gather and analyze large data volumes. Payment providers [52] process high-velocity data streams to perform transactions and prevent faults. Internet services like Spotify [254] and Netflix [113] deploy increasingly complex data processing pipelines to personalize user experiences. To address this environment, cloud vendors like Snowflake [74], Databricks [28], and Amazon [18] offer data processing services that aim for high performance and cost-efficiency [28]. Thus, the need for efficient, robust, and user-friendly data management systems has never been more critical.

1.1 Motivation

Over the last decades, the data management community has dedicated significant efforts towards improving the efficiency and performance of analytical data management systems. Researchers proposed novel processing models [40, 202], data structures [172, 178], leveraged hardware accelerators [44, 83, 185], and revised systems components [177, 205]. A prominent example is query compilation [202], which enables systems to translate queries in workload- and hardware-tailored code at runtime. This work laid the foundation for a new generation of analytical data management systems, e.g., Hyper [162, 202], Hackaton [79], Umbra [203], and MemSQL [216]. These systems focus on in-memory computations and fully exploit modern hardware to reach high performance. However, building such systems is highly complex [28].

At the same time, the data processing environment significantly changed, and workloads, as well as use cases, have become increasingly diverse. Nowadays, data scientists, engineers, and application developers assemble complex and *polyglot data processing pipelines* [9]. These pipelines combine traditional relational operators with user-defined functions (UDFs) that contain specific business logic. Depending on the domain, they are written in dynamic languages such as Java, R, Python, or JavaScript and embed third-party libraries [258]. Additionally, novel use-cases [146, 186] in the area of autonomous driving, fraud detection, and online recommendation perform continuous queries over unbounded, continuously changing, high-

velocity data streams. Thus, modern data processing systems must efficiently support a diverse environment of complex workloads that process large data volumes at high velocity.

Even though general-purpose data processing platforms, like Flink [50] or Spark [285], support this diverse environment, recent research has shown that these systems deliver suboptimal performance [192, 287, 294]. In particular, these systems have not yet adopted many of the previously mentioned techniques for efficient query processing from analytical data management systems. Instead, they follow hardware-oblivious architectures and use managed runtimes to abstract from hardware details. To reach high performance, they focus on horizontal scalability and distribute work across large compute clusters [93]. As a result, these systems neglect single-node efficiency and cannot fully utilize the resources of modern hardware [192]. Thus, users either have to scale out computations across large compute clusters to mitigate performance bottlenecks [93], which increases complexities and costs, or they have to avoid features like UDFs [94, 141, 174], which decreases usability.

Motivated by the above challenges and requirements, the goal of this thesis is to propose techniques to enable *efficient and robust data processing in diverse data processing environments*. In particular, we leverage query compilation as a foundational concept for efficient data processing and investigate its application to a broad set of novel workloads and use cases. Throughout this thesis, we revisit design decisions of general-purpose data processing systems and propose architectural changes to mitigate the bottlenecks of current systems. To this end, we propose individual building blocks that increase the efficiency, flexibility, and maintainability of state-of-the-art systems. Furthermore, we use NebulaStream [289] as a representative data processing system and a target environment for our contributions to demonstrate the practicality of our solutions. In particular, we tackle the previously mentioned performance limitations of current systems in four steps. First, we investigate data processing use cases that analyze high-velocity data streams. We adapt query compilation to the unique characteristics of these workloads and mitigate the performance limitations of current stream processing systems. Second, we investigate polyglot data processing workloads, which combine relational operators with UDFs. We extend our query compiler with support for these workloads and provide novel holistic optimizations that mitigate the performance bottlenecks of state-of-the-art systems. Third, we propose a novel query compilation framework to improve the maintainability of compilation-based execution engines without sacrificing performance. Our framework provides an easy-to-use implementation interface and specialized compilation backends to achieve high performance across diverse workloads. Finally, we integrate our individual contributions in NebulaStream and demonstrate their practical impact.

In summary, this thesis proposes a hardware-conscious architecture for modern data processing systems that target diverse environments beyond traditional relational systems. Our solutions mitigate the bottlenecks of current systems, increase hardware utilization, and achieve significant performance gains while keeping the system's complexity in check. Throughout this thesis, we propose individual building blocks that focus on specific research challenges, e.g., efficient stream processing, support for polyglot queries, and engineering productivity. In combination, our contributions lay the foundation for a new generation of data processing systems, like NebulaStream. As a result, vendors of data processing platforms can leverage our work to increase efficiency and reduce deployment as well as engineering costs.

1.2 Research Problems and Contributions

Modern data processing systems have to target an increasingly diverse environment. These systems have to support polyglot data processing pipelines that combine relational operators with custom UDFs in high-level languages. Additionally, they must efficiently process large data volumes at high velocity. At the same time, they have to keep the engineering and maintenance effort in check. To address these requirements, we propose throughout this thesis fundamental building blocks for the architecture of modern data processing engines. In the following, we discuss the research problems addressed in this thesis and outline our contributions in detail.

1.2.1 Efficient Execution of Continuous Queries

State-of-the-art stream processing systems (SPSs) such as Flink [50] and Spark-Streaming [286] execute long-running continuous queries over high-velocity data streams. These systems are designed for the cloud, leverage managed runtimes like the Java Virtual Machine to abstract from hardware details, and follow shared-nothing architectures to scale out computations across large compute clusters. These design decisions enable virtually unlimited scalability and support high-velocity data streams. However, recent research revealed that these design choices prevent current SPSs from fully utilizing the hardware resources of individual compute nodes [287, 294]. By eliminating these bottlenecks using hand-written implementations of stream processing queries, Zeuch et al. [287] showed that significant performance improvements are possible. In this thesis, we investigate these limitations and propose a novel execution strategy for stream processing queries that reaches the performance of hand-written implementations. In particular, we address the following two problems:

Problem 1: Low Resource Utilization
Current SPSs rely on *query interpretation*, *managed runtimes*, and *partition-based parallelization*, which introduces significant query execution overhead.

Even though state-of-the-art SPSs are highly optimized distributed data processing systems, they neglect single-node performance. Recent work [287, 294] identified three main reasons for their low resource utilization: First, these systems cause many instruction cache misses because they use an interpretation-based processing model. Second, they cause many data cache misses because they rely on managed runtimes. Third, they utilize sub-optimal parallelization strategies on single nodes because they optimize for scale-out environments. To address this limitation, modern SPSs must improve code efficiency and avoid synchronization overhead to fully utilize modern hardware.

Problem 2: Changing Data Characteristics
Current SPSs optimize and deploy queries once and neglect *dynamically changing stream characteristics* at runtime, which may impact execution performance.

Even though state-of-the-art SPSs target long-running queries, they optimize and deploy individual queries only once. Thus, they can not react to changing characteristics of the

underlying data stream at runtime, e.g., a fluctuating ingestion rate, a changing number of distinct values, or a changing data distribution of grouping keys. These changes may impact the efficiency of the selected execution plan and reduce the execution performance. To address this problem, modern SPSs require adaptive execution strategies for stream processing workloads, which detect changes in the underlying data characteristics, and adapt the execution plan at runtime, e.g., by reordering operators or selecting specific algorithms.

Solution 1: Adaptive Query Compilation

We extend *query compilation* for the unique requirements of stream processing workloads and introduce a feedback loop to react to changing data characteristics.

In this thesis, we introduce adaptive query compilation as a novel query execution strategy for SPSs that addresses previously stated limitations of state-of-the-art SPSs. Adaptive query compilation combines the generality and ease of use of SPSs with the efficient hardware utilization of hand-written code. In particular, we make three fundamental contributions:

First, we extend traditional query compilation to the unique semantics of stream processing queries. Data streams are conceptually unbound and need to be discretized into finite windows. In contrast to relational aggregations and joins, windowing semantics are extremely diverse and cover different window types (e.g., tumbling and sliding windows), window measures (e.g., time-based and count-based windows), and window functions (e.g., aggregations). Our query compilation approach provides optimized code generation templates that take these semantics into account and produce efficient code. Second, we establish a feedback loop between code generation and execution. To this end, we combine non-invasive hardware-performance counters to detect changes in data characteristics with specialized profiling code to collect fine-grained statistics at runtime. Based on this profiling information, our execution engine re-optimizes queries adaptively at run time. Third, we apply task-based parallelism and introduce parallelization-aware operators that efficiently utilize multi-core systems. In contrast to relational operators, stateful stream processing operators are order sensitive. To address this requirement, we introduce a lightweight coordination protocol that ensures correctness.

Overall, our adaptive query compilation approach achieves significantly higher throughput than state-of-the-art SPSs, reaches the performance of hand-written code, and lays the foundation for the execution engine of NebulaStream.

1.2.2 Efficient Execution of Polyglot Queries

Polyglot queries extend the relational algebra and combine relational operations with UDFs. UDFs enable users to express arbitrary business logic in their preferred programming language [208], to leverage 3rd-party libraries [258], and to increase modularity and testability [25]. Today, many data processing engines support such polyglot queries [50, 238, 285], e.g., in the form of Java, Python, or JavaScript UDFs. However, their support is often limited to a restricted set of languages and comes with a high performance penalty compared to traditional relational queries [94, 141, 174]. To this end, we investigate in this thesis the following two problems of current execution strategies of polyglot queries.

Problem 3: Invocation Costs of External Runtime's
Current data processing systems support polyglot queries via *3rd-party language runtimes*, causing a significant performance overhead.

Most state-of-the-art data processing systems, e.g., MonetDB [230], Postgres [226], Exasol [189], Impala [278], or Flink [50], embed language runtimes like the JVM or the Python Interpreter to support polyglot queries. For each record, they invoke the runtime, materialize intermediate results, and convert data types, which accumulates to a significant performance overhead. This overhead originates from the fundamental impedance mismatch between the declarative paradigm of SQL and the imperative paradigm of UDFs. As a result, database experts often recommend avoiding the use of polyglot UDFs whenever possible [94, 141, 174]. To address this limitation, modern data processing systems have to bridge the gap between build-in and UDF-based operators. In particular, they require a unified representation, which is agnostic of the origin of operators and enables the efficient execution of polyglot queries.

Problem 4: Optimizing Polyglot Queries
Current data processing systems cannot perform *optimizations* across relational and UDF-based operators, which leads to suboptimal execution plans.

State-of-the-art data processing systems treat UDF-based operators as black-boxes [241]. As a result, the processing logic is scattered between the native query execution engine and the language runtime, adding a level of indirection and preventing query optimization. Consequently, polyglot queries often lead to suboptimal query execution plans. To address this limitation, a polyglot query execution engine has to handle built-in and UDF-based operators on the same unified representation. Based on this, an optimizer could eliminate operator boundaries and leverage query optimizations for polyglot queries.

Solution 2: Polyglot Query Compilation
We introduce a *query compilation approach for polyglot queries* that unifies the execution of relational operators and UDFs in a single engine.

In this thesis, we introduce a novel approach for efficiently executing polyglot queries. We extend query compilation for polyglot queries and unify the execution of relational operators and UDFs in a single engine. In particular, we make the following three contributions:

First, we introduce a unified intermediate representation (IR) that supports relational built-in operators and UDFs across different programming languages. Our IR models the unique properties of diverse polyglot operators and the data exchange among them in a common representation. Furthermore, it represents queries across different levels of abstraction to support optimizations, i.e., from the initial logical query plan down to the final machine code. Second, we leverage this IR to apply traditional and new query optimizations to polyglot queries in a holistic and operator-agnostic manner. This enables our execution engine to specialize query processing to the specific requirements of polyglot queries, e.g., by applying optimizations such as operator fusion [202] or predication [239]. Third, we utilize query compilation to translate polyglot queries into highly efficient code fragments. To this end, we

leverage adaptive execution to specialize the execution of UDFs written in dynamic languages like JavaScript or Python towards data characteristics at runtime.

Overall, our approach significantly reduces the overhead of polyglot queries and outperforms state-of-the-art systems by up to one order of magnitude. Furthermore, it lays the foundation for the support of polyglot queries in NebulaStream.

1.2.3 Efficient Development of High-Performance Execution Engines

Engineering high-performance query execution engines is a challenging task. To this end, it is crucial to choose a suitable query execution engine architecture. Over the last decade, vectorized query interpretation [40] and query compilation [202] have emerged as prevailing architectures for high-performance query execution engines. Vectorized query interpretation passes vectors of records between precompiled operators that can be developed in ordinary imperative code. In contrast, query compilation translates queries into specialized code at runtime. This provides excellent performance but introduces a high system complexity, making the engine hard to build, debug, and maintain [28, 218]. To this end, we investigate in this thesis the following two limitations of current query compilation approaches.

Problem 5: Managing High Engineering Effort

Building, debugging, and maintaining high-performance query compilers is very challenging and requires significant developer time and costs.

As with any other software artifact, developing, debugging, and maintaining query execution engines requires developer time and costs [28, 218]. In contrast to interpretation-based engines, query compilers generate operator code after query submission, i.e., at runtime. This introduces a gap between implementation and execution, which makes engines hard to build, debug, and maintain. To overcome this challenge, a compilation-based engine should hide code generation details and focus on providing high productivity. However, at the same time, they have to produce efficient code across a wide range of workloads.

Problem 6: Navigating a Large Design Spaces

Engineers of query compilers have to navigate a *large design space* of different architectures and different trade-offs between performance and maintainability.

Over the last years, research introduced specialized query compilers for short-running queries [103, 162], stream processing [31, 125, 265], or user-defined functions [71, 128]. These compilers follow different software architectures and trade-off between compilation time, execution performance, and developer experience. However, there is currently no query compiler available that is suitable for a wide range of diverse workloads. As a result, engineers must develop compilation-based engines from scratch and reinvent solutions for reoccurring tasks like the integration of compiler frameworks [151]. Thus, the development of compilation-based execution engines requires skills from multiple domains, which makes hiring complex [218]

Solution 3: Maintainable Query Compilation

We introduce a novel approach for developing query execution engines that provide the *advantages of query compilation without sacrificing developer productivity.*

In this thesis, we introduce a novel framework for developing query compilation-based query execution engines. This framework combines an interpretation-based programming model to ensure high productivity with a flexible just-in-time (JIT) compiler, which translates operators into efficient code. To address the limitation above, we make the following contributions:

First, we introduce a generic interface for implementing diverse data processing operators. This avoids the complexity of code generation and enables engineers to write imperative C++ code that is easy to develop, debug, and maintain. Second, we provide a novel JIT compiler that uses tracing to translate imperative operators to an intermediate representation and uses one of multiple compilation backends to produce efficient code. This enables the compiler to specialize query execution towards particular workload requirements, e.g., minimizing startup latency or maximizing execution performance. Third, we introduce a common runtime across all compilation backends. This runtime provides shared data structures and enables engineers to reuse common components across operators. Furthermore, this simplifies the implementation of backends and improves the testability.

Overall, our query compilation framework hides the complexity of high-performance query execution engines. It compiles queries to specialized and efficient code without sacrificing the productivity of engineers. Furthermore, the framework is agnostic to specific workloads, supports batch, streaming, and polyglot queries, and lays the foundation for NebulaStream's compilation-based execution engine.

1.3 Impact of Thesis Contributions

During the course of our research, we have made the following contributions.

Research Publications. The core results of this thesis resulted in the following peer-reviewed publications that have been published at international top-tier venues:

1. **Philipp M. Grulich, Sebastian Breß, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl, Volker Markl**: *Grizzly: Efficient Stream Processing Through Adaptive Query Compilation.* ACM SIGMOD, 2020.

2. **Philipp M. Grulich, Steffen Zeuch, Volker Markl**: *Babelfish: Efficient Execution of Polyglot Queries.* PVLDB, 2021.

3. **Philipp M. Grulich, Aljoscha Lepping, Dwi Nugroho, Varun Pandey, Bonaventura Del Monte, Steffen Zeuch, Volker Markl**: *Towards Unifying Query Interpretation and Compilation.* CIDR, 2023.

4. **Philipp M. Grulich, Aljoscha Lepping, Dwi Nugroho, Varun Pandey, Bonaventura Del Monte, Steffen Zeuch, Volker Markl**: *Query Compilation without Regrets.* Under Submission.

Open Source Contributions. We have released the following software artifacts for the contributions of this thesis under Apache License 2.0.

- https://github.com/TU-Berlin-DIMA/grizzly-prototype: This repository contains the source code of our Grizzly prototype and illustrates our adaptive query-compilation approach [125]. To this end, it provides documentation, data generators, and benchmarks.
- https://github.com/TU-Berlin-DIMA/babelfish: This repository contains our Babelfish engine for the efficient execution of polyglot queries [128]. We provide documentation, examples, and references for the benchmarks.
- https://github.com/nebulastream/nebulastream[1]: This repository contains the NebulaStream system and provides the code for our Nautilus query compiler.

1.4 Structure of the Thesis

This thesis is structured as follows.

Chapter 2: Query Compilation. In this chapter, we introduce query compilation and provide the necessary background knowledge that serves as a basis for our thesis. Furthermore, we describe the architecture of NebulaStream's query compiler as it is the result of our contributions and demonstrates their practicality.

Chapter 3: Query Compilation for Stream Processing. In this chapter, we propose a query compilation approach for stream processing queries. In particular, we address the unique characteristics of stream processing workloads and propose adaptive optimizations for long-running queries. Based on this architecture, we show in our evaluation that adaptive query compilation can fully leverage hardware resources and results in significantly higher performance in comparison to state-of-the-art SPSs.

Chapter 4: Query Compilation for Polyglot Queries. In this chapter, we describe the acceleration of polyglot data processing workloads. We propose a novel compilation-based engine that holistically optimizes polyglot queries that involve relational operators and UDFs in Python, Java, and JavaScript. This enables our engine to eliminate operator boundaries and to mitigate the performance limitations of current systems.

Chapter 5: Query Compilation Without Regrets. In this chapter, we introduce a novel approach for developing query execution engines that provides the advantages of query compilation without sacrificing developer productivity. To this end, we introduce a framework that decouples the implementation of operators from code generation. As a result, our framework can reach a high performance across a variety of data processing workloads.

Chapter 6: Additional Contributions. In this chapter, we describe further related research contributions, which have been made while working on this thesis but are not covered in other chapters.

Chapter 7: Conclusion. In this chapter, we conclude the thesis by summarizing our contributions and providing an outlook for future work.

[1]To access the source code of NebulaStream, we require a registration at http://nebula.stream.

2

Query Compilation

Over the last decade, query compilation has emerged as a prevalent query execution approach in modern database systems [6, 102, 166, 195, 202, 203, 215, 217, 278]. At the same time, various compiler architectures and code generation techniques have been proposed. In this chapter, we provide an introduction to query compilation and introduce the necessary background that serves as a basis for this thesis. First, we provide an overview of query compilation (see Section 2.1). Based on this, we introduce design aspects for the architecture of compilation-based execution engines (see Section 2.2). Finally, we present the design of NebulaStream's query compiler, which is a foundation for the remainder of this thesis (see Section 2.3).

2.1 Query Compilation at a Glance

In the realm of query execution engine architectures, we can distinguish between two fundamental query execution strategies, *query interpretation* [116] and *query compilation* [202]. Early data management systems like System R generated machine code directly from query plans [55]. They translated relational operators into predefined machine-code templates to avoid the overhead of parsing, type checking, and access path selection at runtime. Nevertheless, systems of this era were mainly limited by the IO-bottleneck of hard disks. As a result, query compilation got replaced by interpretation-based systems [116], which were easier to develop and maintain [19]. With the adoption of main-memory systems in the last decade, query compilation gained popularity again. Systems, such as Hyper [202], and Hekaton [79] translate queries into efficient code to reduce interpretation overhead and maximize performance.

Figure 2.1 provides a high-level overview of the architecture of a data management system and illustrates both query interpretation and compilation. Independent of the query execution strategy, systems initially parse input queries ① and perform optimizations ② to derive a physical query plan. Based on the physical query plan, query interpretation-based engines instantiate a set of predefined executable operators ③. These operators either correspond directly to relational operators, e. g. selections or joins, or to a set of low-level plan operators [184]. At query runtime, the execution engine passes data via virtual function calls

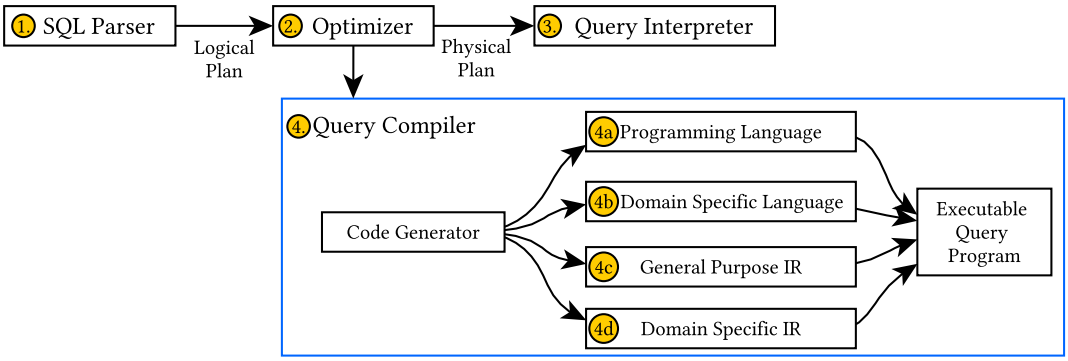


Figure 2.1: Architecture of a query execution engine, which uses either query interpretation or query compilation.

between the individual operator. This provides high flexibility and debuggability but also introduces a significant performance overhead [202].

In contrast, query compilation-based engines translate the physical query plan to one or more specialized machine code fragments (4). The query compiler first traverses the query plan and generates for each operator a code fragment in some form of intermediate representation (IR), which captures the operations that an operator executes on the input data. This can either be a programming language (4a), a domain-specific language (4b), a general purpose compiler IR (4c) like LLVM-IR, or a domain-specific IR (4d) like Umbra-IR [157] or Flounder-IR [103]. Using these IRs, query compilers can perform further optimizations, e.g., operator fusion or vectorization, before they finally generate executable machine code. Thus they can specialize the query execution for specific workloads, data characteristics, or hardware.

Overall, a query compiler has to make several design decisions: 1) *How to translate operators to an IR?* 2) *How to represent data processing semantics in an IR?* 3) *How to translate the IR to machine code?* 4) *How to integrate compiled queries in the execution engine?* Over the last years, research and industry proposed various query compiler architectures to explore this design space. These optimize for latency [103, 157], target specific hardware like GPUs [46, 104, 219], or focus on different workloads, like stream processing [31, 125, 265], machine-learning [75], or the execution of UDFs [71]. Depending on these decisions, we can compare query compiler architectures using the following metrics as proposed by Gruber et al. [121]:

M1 Throughput: For a query compiler, throughput is influenced by the quality of the generated code. If the code quality is high, processing becomes more efficient, and more tuples can be processed per second. Thus, maximizing throughput is particularly crucial for long-running batch or streaming workloads. To achieve this goal, query compilers can leverage mature compiler frameworks like LLVM to apply advanced compiler optimization passes [130]. However, additional optimizations also increase compilation latency [162].

M2 Latency: For a query compiler, latency is defined by the time the compiler requires to translate the physical query plan to executable machine code. Depending on the query complexity, the selected compilation approach, and the number of performed optimizations, the compilation latencies vary between milliseconds and multiple seconds [162]. The compilation latency can impact query execution time significantly, and systems that target short-running queries on small data require low-latency query compilers [103, 157].

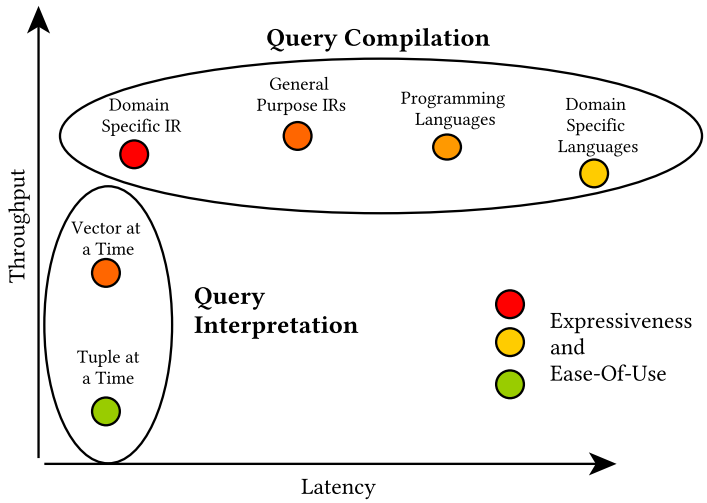


Figure 2.2: High-level sketch of the design space for the architecture of query compilers.

M3 Expressiveness: The intermediate representation of a query compiler captures the semantics of data processing operators at a specific abstraction level. We can differentiate between declarative domain-specific languages [129, 259], which resemble primitive data processing operations and enable optimizations, and low-level IRs [103, 157], which cover the architectural hardware-specific aspects, e.g., memory models or specific constraints. Some query compilers also leverage multiple IRs to optimize across different levels of abstraction.

M4 Ease of use: As with any other software artifact, building, debugging, and maintaining query compilers requires developer time and costs [28]. Depending on the query compilation approach, the abstraction level of its IR, and the maturity of the compiler, the ease of use and the costs of developing new operators vary [198]. Thus a high ease of use for the development of compilation-based execution engines is required to increase developer productivity.

Overall, a query compiler has to trade-of between these individual design dimensions. In the remainder of this chapter, we will explore the individual aspects of this design space.

2.2 Design Space of Query Compilers

Query compilers are complex software components that can be characterized by the design decisions discussed in Section 2.1. These target specific workload requirements, determine the expected runtime performance and have to fit to the architecture of the whole query execution engine. Figure 2.2 represents this design space and highlights the trade-offs between throughput, latency, and ease of use. Depending on the execution strategy, as well as the architecture of the code generator, the resulting system has a different performance profile.

In this section, we introduce these individual design aspects in detail. To this end, we first discuss the prevailing query execution strategies of modern data processing engines and investigate how they influence the design of query compilers in Section 2.2.1. Based on this, we discuss approaches for the translation of operators to code in Section 2.2.2 and common forms of intermediate representation in Section 2.2.3. Finally, we discuss strategies to translate the IR in an executable program and how it interacts with a query execution engine in Section 2.2.4.

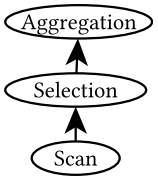
2.2.1 Query Execution Strategy

The query execution strategy of an execution engine determines how a physical query plan is evaluated. Over the last decades, several strategies have been proposed which make different trade-offs between performance and flexibility. We can differentiate between the inter-operator execution strategy, i.e., *pull* and *push-based* execution, the intra-operator execution strategy, i.e., a *tuple-at-a-time*, *operator-at-a-time*, *vectorized*, and *data-centric*, and different parallelization strategies. Depending on the combination of these strategies, the resulting engine may either fit more towards a query interpreter or a query compilation-based approach. In the following, we detail these aspects and indicate the individual trade-offs.

2.2.1.1 Inter-Operator Execution

Each operator in a physical query plan implements a particular step of a query. These operators either directly map to one of the operations of relational algebra or represent specific tasks that are required for query execution [107]. For instance, consider the following query and its physical query plan:

```
Select sum(u.value)
From user u
Where u.id == 42
```

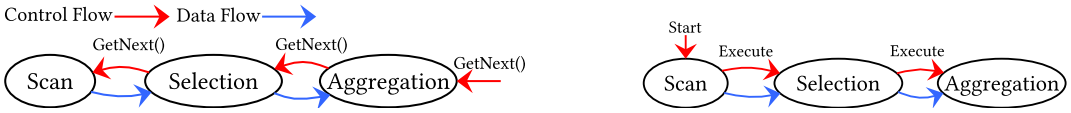


The physical query plan consists of the three operators, i.e., the `scan` accesses the base relation and loads individual tuples, the `selection` selects tuples depending on its predicate `u.id == 42`, and the `aggregation` performs a sum aggregation `sum(u.value)`. To execute this query, we can differentiate between *pull*- and *push*-based strategies illustrated in Figure 2.3.

Pull-based query execution: This strategy evaluates queries from the root, i.e., the aggregation in our case, to the leaf, i.e., the scan on the base relations. Each operator invokes its operator and *pulls* the next result tuple. Finally, the scan operator is called, accesses the base relation, and loads the next tuple. Traditionally this model is implemented through an iterator interface [116], which defines three methods: `Open` initializes a physical operator and its state. `GetNext` returns the next tuple of the result and adjusts the state of the operator. `Close` terminates the iteration if the child operator has retrieved all tuples it requests.

Push-based query execution: In this strategy, the data flow between operators is initiated at the base relation (the scan), which loads tuples and pushes them towards its parent operator [204], i.e., to the selection in our example. Each operator processes the data and passes it onward to the next operator. Stateful operators, e.g., aggregations or joins, must materialize the intermediate state until they are notified that all data was processed. To this end, operators must implement an `Execute` method, which receives the next tuple and reacts to specific events to coordinate progress, e.g., if an operator wants to terminate [231].

In general, the pull-based query execution strategy can be described as a consumer-driven model where operators are only invoked when the consumer (root operator) demands a new result tuple [249]. This has the advantage that tuples are only loaded and produced if needed, e.g., a `limit` operator can directly control how many tuples are produced. However, the pull-based model introduces a complex bi-directorial control- and data-flow between operators.



(a) Pull Model. **(b)** Push Model.

Figure 2.3: Illustration of the **pull-(a)** and **push-(b)** query execution model for a scan, select, aggregate query.

This causes frequent virtual function calls and context switches between operators, which may lead to a high query processing overhead [202].

In contrast, the push-based query execution strategy is a producer-driven model, as the base operators start processing data without waiting for demand from consumer operators [249]. This model aligns the control- and data-flow between operators, i.e., from the leaf to the root operator. This simplifies the data path and can improve cache utilization [204]. Furthermore, the push-based strategy fits well to a task-based parallelization scheme, which is crucial to exploit modern multi-core hardware efficiently [176, 290], and naturally supports continuous query processing, where operators may produce data continuously for multiple consumers [138, 204]. However, the push-based model requires a more sophisticated scheduling system to distribute work between operators, to avoid overloads, and to signal progress [229].

Most traditional data management systems, like MS SQL Server [116], PostgreSQL [226], MonetDB [40], follow the original iterator model and leverage a pull-based execution strategy. This strategy enabled high flexibility and sufficient performance for disk-based systems. However, with the increasing demand for high-performance analytical data processing and the increasing adoption of modern main memory data processing systems, the pull-based query execution strategy became a significant performance bottleneck [202]. To mitigate this bottleneck, most recent data executions engines favored a push-based execution strategy, e. g., Hyper [202], Umbra [203], Volex [221], and DuckDB [229].

2.2.1.2 Intra-Operator Execution

Orthogonal to the execution strategy between operators, we can also differentiate between different intra-operator query execution strategies. These strategies define the granularity at which tuples are processed within operators and result in different performance characteristics. In the following, we introduce the five primary intra-operator execution strategies illustrated in Listing 1, i.e., *tuple-at-a-time*, *vector-at-a-time*, *operator-at-a-time*, *data-centric*, and *hybrid*.

Tuple-at-a-Time: In this query execution strategy (Listing 1a), each operator processes one tuple at a time and passes it directly to the next operator. Tuple-at-a-Time processing is traditionally combined with the pull model in interpretation-based execution engines [116]. It allows a simple and flexible operator implementation but causes significant processing overhead due to the frequent virtual function calls between operators [11].

Operator-at-a-Time: In contrast, to the tuple-at-a-time strategy, the operator-at-a-time approach increases the processing granularity. Each operator consumes an entire column of values, materializes its result, and passes them to the next operator. This model was pioneered by MonetDB [40] to mitigate the function call overhead of the tuple-at-a-time

```
class Scan {
 void execute(Buffer b){
  for (record: b)
   child.execute(record);
}}

class Selection {
 void execute(Tuple t){
  if(t.id == 42)
   child.execute(t);
}}

class Aggregation {
 int sum;
 void execute(Tuple t){
  sum += t.value;
}}
```

(a) Tuple-at-a-Time

```
class Scan {
 void execute(Buffer b){
  for (vector: b)
   child.execute(vector);
}}

class Selection {
 void execute(Vector v){
  for(i = 0; i< v.size; i++)
   v[i].p = v[i].id == 42;
  child.execute(v);
}}

class Aggregation {
 int sum;
 void execute(Vector v){
  for(i = 0; i< v.size; i++)
   sum += v[i].p * v[i].value;
}}
```

(b) Vector-at-a-Time

```
class Pipeline {
 int sum;
 void execute(Buffer b){
  for (tuple: b)
   if(tuple.id == 42)
    sum += tuple.value;
}}
```

(c) Data-Centric

```
class Pipeline {
 int sum;
 void execute(Buffer b){
  for (c: b)
   for(i = 0; i< c.size; i++)
    if (c[i].id == 42)
     idx[j++] = i;
   for(k = 0; k < j; k++)
     sum += c[idx[k]].value;
}}
```

(d) Hybrid

Listing 1: Pseudo-code that illustrates the different intra-operator execution strategies following the push-based model: **a) tuple-at-a-time**, **b) vector-at-a-time**, **c) data-centric**, **d) hybrid**.

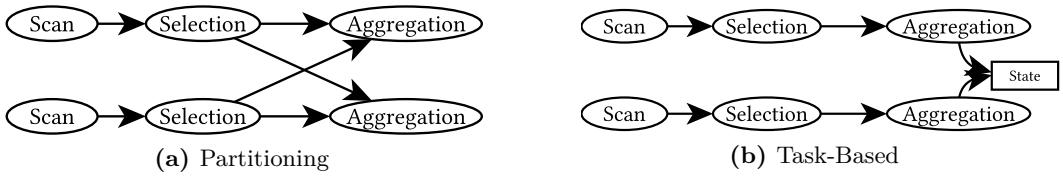
strategy. However, it enforces the full materialization of intermediate results, which can also cause significant overhead [202].

Vector-at-a-Time: In contrast, to the operator-at-a-time strategy, the vector-at-a-time approach (Listing 1b) operates on chunks of data, so-called vectors that fit into the CPU cache [40]. This also reduces the frequency of function calls between operators, improves cache utilization as vectors usually fit in the CPU caches, and enables the exploitation of SIMD operations to improve performance. However, to reach optimal performance, engineers have to provide optimized compute kernels that provide vectorized implementations of all data processing operations [190].

Data-Centric: This query execution strategy (Listing 1c) is commonly used by compilation-based data execution engines and fuses multiple operators to one combined fragment/pipeline [202]. This pipeline receives a chunk of tuples, similar to the vector-at-a-time model, but performs multiple operators in one tight loop. This avoids data materialization, allows values to stay in CPU registers, and enables efficient query processing. However, it is challenging to leverage SIMD operation as this strategy operates only on individual scalar values.

Hybrid: This strategy (Listing 1d) combines the benefit of a data-centric and vector-at-a-time execution strategy [72, 194]. It introduces *subpipelines* that fuses or splits operators to minimize intermediate materialization while also enabling SIMD processing opportunities. Similar to the vector-at-a-time strategy, it collects intermediate values in cache-sized buffers, which are processed in tight nested loops. However, each loop can contain multiple operators.

Overall, we can observe that state-of-the-art data processing systems follow two directions. Recent interpretation-based systems, e.g., DuckDB [231], Velox [221], and Photon [28], follow the vector-at-a-time model. Mature compilation-based systems, e.g., Hyper [202], Umbra [203], and SingleStore [217], apply the data-centric model. In contrast, hybrid models are mainly explored by research prototypes, e.g., Tuplewise [70, 71] and NoisePage [194].



(a) Partitioning **(b)** Task-Based

Figure 2.4: Illustration of **(a) partitioning** and **(b) task-based** parallelism.

2.2.1.3 Parallelization

In addition to the execution strategy, query execution engines can also be characterized by their parallelization strategy. This particularly becomes crucial for data processing on modern multi-core servers that have up to 100s of cores across different NUMA-regions [24]. For these systems, it is critical to distribute the workloads across multiple threads while avoiding synchronization overhead. In general, we can differentiate between parallelization strategies where operators are either parallelism *aware* or *unaware*, illustrated in Figure 2.4.

Traditionally, database systems adopted the Volcano [116] model to parallelize query execution, illustrated in Figure 2.4a. This model introduces a special exchange operator that partitions tuples between parallel operator instances. As a result, data processing operators can remain *parallelism unaware*, which simplifies their implementation. However, the distribution of partitions is static, which can cause an imbalance at query execution time.

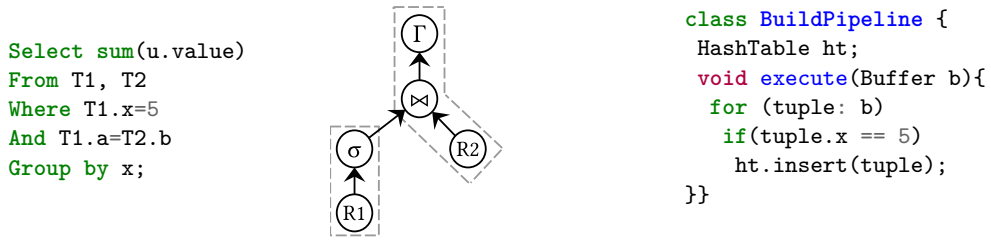
In contrast, task-based or morsel-driven parallelism, illustrated in Figure 2.4b, introduces a scheduling mechanism that dynamically routes chunks of tuples between pipelines of concurrently executed operators [176, 290]. To this end, operators are *parallelism aware* and concurrently operate on a shared state. This requires efficient mechanisms to avoid synchronization overhead, e.g., local preaggregation or lock-free hash tables.

Across state-of-the-art data processing systems, we can observe that most engines follow a task-based approach as it was shown to scale efficiently on modern multi-core hardware [176, 290]. Furthermore, task-based parallelization was leveraged to enable query processing on heterogeneous hardware [83, 199]

2.2.2 Code Generation

Code generation is an essential phase in compilation-based execution engines. It translates the operators of a physical query plan into code fragments in some form of intermediate representation. Code generators aim to generate efficient code by fusing operators, avoiding intermediate results, and specializing the code towards hardware or workloads characteristics.

To translate operators into code fragments, most query compilers adopted the *produce/-consume* model [202]. This approach segments the query plan in pipelines whenever a materialization of intermediate results is required. It defines operators that materialize state as *pipeline-breakers*, e.g., Aggregations or Joins. All operators inside a pipeline are fused together and result in one combined code fragment, which follows a data-centric execution strategy. Thus it performs a single pass over the data such that data stays in CPU registers. In combination with a task-based parallelization strategy, pipelines represent atomic instances of parallelization [176]. As a result, pipelines can be executed independently and concurrently with other pipelines, but execution within a pipeline remains sequential.



(a) Query **(b)** Pipelined Plan **(c)** Generated Code

Figure 2.5: Illustration of produce/consume model for query compilation.

Figure 2.5 illustrates how an example query is translated into pipelines. The query performs a join between R1 and R2 and terminates with an aggregation. As building the hash table for the Join between R1 and R2 is a pipeline breaker, the produce/consume model creates one build pipeline on table R1 and one probe pipeline on table R2. To translate the operators into code fragments, Neumann [202] introduced two functions `produce` and `consume`. From the root operator, the query compiler recursively invokes `produce` to ask the operator to produce its result tuples. The base operator generates code for the `scan` and calls the `consume` function on its parents. In the consume function, each operator inserts its processing code, e.g., the `if(x == 5)` for the selection in our example and the `ht.insert` in the join build.

Over the last years, research proposed different variations of the produce/consume model. Crotty et al. [71] and Menon et al. [194] introduced *sub pipelines* to generate code that follows a hybrid execution strategy. This enables vectorization and improves cache utilization. Kersten et al. [157], introduce programming abstractions to reduce the complexity of code generation. In particular, he reduced the operator interface to a single `consume` function and proposed a toolbox of predefined data structures and primitives to generate operator code. Breß et al. [45] introduced the notation of pipeline programs that specialize code generation toward specific accelerators, e.g., GPUs or FPGAs. Depending on the target hardware, a pipeline program is translated into a specialized code variant that can exploit specific hardware characteristics. An additional line of work, leveraged high-level compliation frameworks like LMS [237] to hide code generation [250, 259]. For instance, Tahboub et al. [259] replaced the produce/consume model with a callback-based interface. This enables engineers to implement operators using a tuple-at-a-time [116] execution strategy similar to an interpretation-based engine.

Overall, the code generation approach of a query compiler determines how engineers can develop individual data processing operators. It defines an interface to model and connect operators and provides reusable primitives and data structure. Thus, easy-to-understand and expressive abstractions are required to reach a high ease of use and maintainability.

2.2.3 Code Representation

One key aspect in designing a query compiler is selecting the intermediate representation. This IR serves as the target for code generation and represents the semantics of physical operators. Over the last years, research proposed a variety of intermediate representations with different abstraction levels, performance goals, and target environments. Overall, we can differentiate four categories, illustrated in Figure 2.6, i.e., *programming languages*, *domain specific languages*, *general purpose IRs*, and *domain specific IRs*.

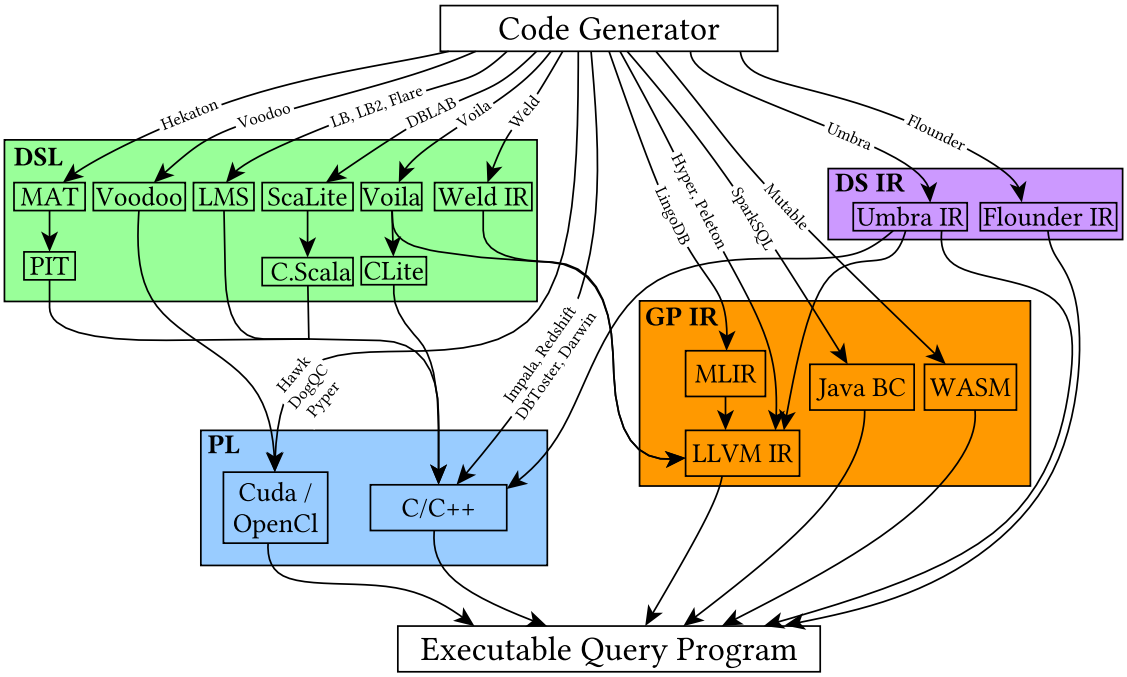


Figure 2.6: Overview of different intermediate representations for query compilers adopted from Gruber et al. [121]. Domain Specific Languages (DSL), Domain Specific IR (DS IR), General Purpose IR (GP IR), and Programming Languages (PL).

2.2.3.1 Programming Languages

The most direct approach for the implementation of a query compiler is the generation of code in a common programming language like C or C++ [10, 18]. This approach allows the code generator to produce code in the implementation language of the execution engine, making the generated code comprehensible and easy to debug for engineers [18]. After code generation, the generated code is usually compiled and optimized with a standard compiler like Clang or GCC, which can produce highly efficient machine code. Additionally, this approach enables seamless interaction between the generated code and runtime components, e.g., to utilize specific data structures and to integrate with the standard language library.

Furthermore, this approach is commonly used to generate code for accelerators like GPUs and FPGAs [45, 104, 219]. These query compilers produce CUDA or OpenCL code, which is optimized for the specific target device. For instance, HAWK [45] generates OpenCL code, which optimizes memory access patterns for specific devices, and DogQC [104] generates Cuda code that mitigates branch divergence by reassigning work to GPU lanes at runtime.

Even though generating code in common programming languages is easy to implement and results in efficient machine code, it has two significant drawbacks. First, compilers for general-purpose programming languages primarily focus on generating optimal code and neglect compilation latency, which can accumulate up to multiple seconds for complex queries [202]. As a result, this approach renders unpractical for data processing systems, which frequently execute short-running queries as the compilation time of queries becomes a major overhead. To mitigate this overhead, Redshift introduces multiple compilation caches [18]. This prevents the compilation of similar queries but introduces additional engineering complexity. Second, the direct generation of code in common programming languages can also prevent workload-specific optimizations. In particular, the compiler, e. g. GCC or Clang, becomes a black box, making

it hard for engineers to reason if, why, and how a specific query was optimized. As a result, most of the recently proposed query compilers neglected this approach.

2.2.3.2 General Purpose IRs

In contrast to generating code in general-purpose programming languages, Neumann [202] proposed to translate operators to a general purpose IRs (GP IRs). Suitable compilation targets are LLVM IR [194, 202], Java Byte Code (JavaBC) [6], or WebAssambly (WASM) [133]. These IRs typically stem from the mid-level stages of general-purpose compilers and eliminate any overhead that stems from the language front end. For instance, Clang is the C++ front end for LLVM IR, while JavaC compiles Java files to Java BC. As a result, compiling these IRs to machine code usually requires only several milliseconds, making this approach more suitable for short-running queries. Furthermore, these IRs offer more control over the generated machine code, which the data processing system can use to optimize specific workloads. Consequently, this approach was adopted by several query compilers over the last years [6, 133, 151, 194, 202].

Despite the advantages of using compiler IRs, this approach also has the following limitations. First, GP IRs like LLVM-IR may still induce substantial overhead on very complex or very short-running queries [157, 162]. Furthermore, GP IRs are rather low-level and necessitate specialized expertise from engineers, e.g., LLVM-IR is often described as a portable form of assembly. In particular, they follow traditional IR structures from compiler research, such as Static Single Assignment (SSA) form, and operate on virtual registers, which renders the IR challenging to comprehend and debug. To address this issue, Jungmair et al. [151] recently proposed the adoption of MLIR [175]. MLIR provides a compiler implementation framework that enables engineers to create a high-level abstraction, which is subsequently lowered to LLVM-IR. However, it remains to be seen whether a compilation framework is a suitable environment for implementing complete data management systems.

2.2.3.3 Domain Specific Languages

Several research projects proposed domain-specific languages (DSLs) to raise the abstraction levels for intermediate representations of query compilers [129, 215, 225, 250, 259]. In contrast to programming languages and compiler IRs, these DSLs model declarative data processing operations that are used across different operators and can enable further optimizations. For instance, Weld [214, 215] proposes an IR that models basic data processing operations, e.g., arithmetic, value assignments, sequential loops, and operations on collections like a hash table. Voodoo [225] offers parallelizable vector operations like `scatter` and `foldsum` that can be optimized for different hardware. In contrast, Voila [129] offers fine-grained primitives like `hash`, `read_pos`, `bucket_insert`, and `bucket_lookup` to describe specific operator details on a common level, e.g., to model different join algorithms. Consequently, selecting a suitable abstraction level for a DSL is challenging. To this end, DBLAB [250] and LingoDB [151] lower queries across multiple levels of DSLs. Each DSL represents the query at a different abstraction level and offers different optimizations. However, these abstractions also introduce indirection, increase complexity, and make individual queries hard to debug.

Finally, all systems lower their DSL to a specific target IR, which can be compiled to machine code. For instance, Hakaton, LegoBase, LB2, and DBLAB, target C/C++, Voodoo

targets OpenCl, and Weld as well as Voila target LLVM IR. Consequently, the specific compilation target limits the performance of a DSL-based query compiler. For example, Tahboub et al. [259] reported compilation times in the order of seconds for DBLAB and LB2, and Excalibur [130] introduces caching for Voila to mitigate compilation overhead. As a result, assessing the benefits of a declarative DSL for the design of a query compiler remains crucial.

2.2.3.4 Domain Specific IRs

Recently several query compilers [103, 157, 195, 203, 217] introduced domain-specific IRs (DS IRs), which can be characterized as an intersection between the previously discussed DSLs and GP IRs from the compiler community. Similar to GP IRs, they introduce a set of instructions, e.g., `add`, `load`, and `store`, to represent the implementation of operators imperatively. To model control and data-flow between instructions, they use common techniques from compiler research, e.g., SSA form and basic-blocks. Similar to DSLs, they also introduce specific domain-specific instructions for data processing. For instance, NoisePage [195] and Umbra [203] define instructions to represent NULL semantics of SQL, to construct iterators over tables, and to operate on indexes and hash tables. However, the abstraction level of DS IRs is usually much lower compared to the previously discussed DSLs, which model high-level operations.

Based on the DS IRs, systems follow different strategies to generate an executable query. NoisePage [195], and SingleStore [217] primarily lower their IR to LLVM to generate highly efficient code and use a bytecode interpreter for short running queries [162]. Flounder [103], directly emits machine code to reduce compilation time at the cost of reduced code efficiency. Umbra [157] follows all three approaches to reach optimal performance for short-running as well as long-running queries.

Even though DS IRs offer a high degree of flexibility and have been shown to enable high execution throughput and low compilation latency, they also induce a high level of complexity. Similar to general-purpose IRs, they represent the implementation of operators at a very low level, which makes debugging the generated code challenging. In particular, the direct generation of machine code in Flounder and Umbra is very challenging as details of target-specific instruction and memory models have to be modeled correctly [121].

2.2.4 Code Execution

In the final phase of a compilation-based query execution engine, the system translates the generated code to an executable program and registers it in the execution engine. To this end, we can differentiate between two components. The *Virtual Machine*, in which the executable query program is executed, and the *Runtime System*, which provides a common infrastructure, e.g., data structures, maintains operator state and interacts with the VM and the host engine.

2.2.4.1 Virtual Machine

The Virtual Machine (VM) is a key component of a compilation-based query execution engine. It receives the generated IR from the code generator, manages its execution, and frees its resources if the query is terminated. To translate the IR into an executable program, we can differentiate two approaches, *eager compilation* and *adaptive compilation with bytecode*

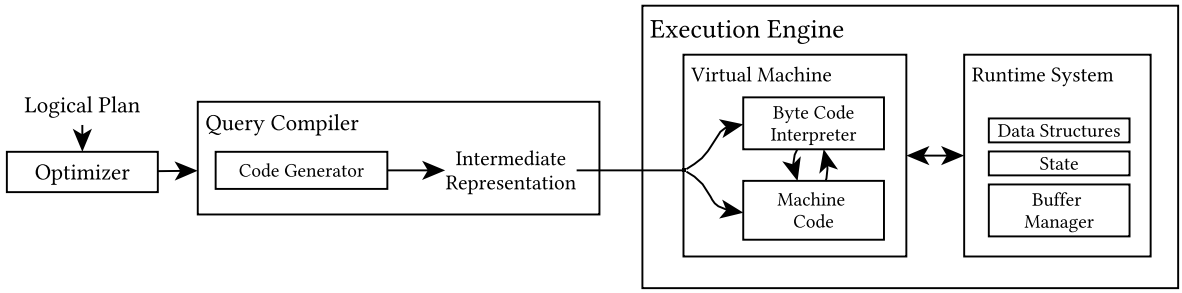


Figure 2.7: Illustration of the execution of compiled queries in compilation-based query execution engines.

interpretation. Traditionally, query compilers eagerly compiled their IR to machine code [202]. Depending on the type of IR, e.g., code in a programming language or a general-purpose compiler IR, this requires a substantial compilation time before the query can process any data. To mitigate this initial latency, Kohn et al. [162] introduced a bytecode interpreter that executes the generated IR without compilation. Only if the system detects that compilation would be beneficial, the VM compiles the query to efficient machine code. While executing a query, the VM interacts with several components of the runtime system, e.g., to access state or emit data. In systems like NoisePage [195] and Umbra [245], the VM also interacts with the optimizer to perform adaptive optimization, e.g., to reorder operators at runtime.

2.2.4.2 Runtime System

The Runtime System provides a common infrastructure for the VM and the generated code. For example, the build-side of a join accesses the operator state, operates on a specific data structure, and allocates pages from the buffer manager to store data.

For query compilers that use general-purpose programming languages as IRs, the runtime system and the generated code are written in the same language. As a result, the generated code can directly include runtime code, which enables the compiler to optimize both holistically, e.g., by performing inlining. In contrast, most systems that use GP IRs like Hyper [202] use function calls to access components of the runtime system from the generated code. This makes it possible to test and compile runtime components before query compilation but also introduces function call overhead. In contrast, Haffner et al. [133] proposed that also runtime code is generated by the query compiler, e. g. all code that interacts with a hash table. This mitigates the overhead of such function calls and enables compiler optimizations. However, it also introduces a high engineering complexity as already implemented and tested data structures from the runtime system can not be used and have to be reimplemented.

Consequently, a query compiler must define an interface between the generated code and the runtime system, which incorporates the architecture of the query compiler, the IR, as well as the runtime.

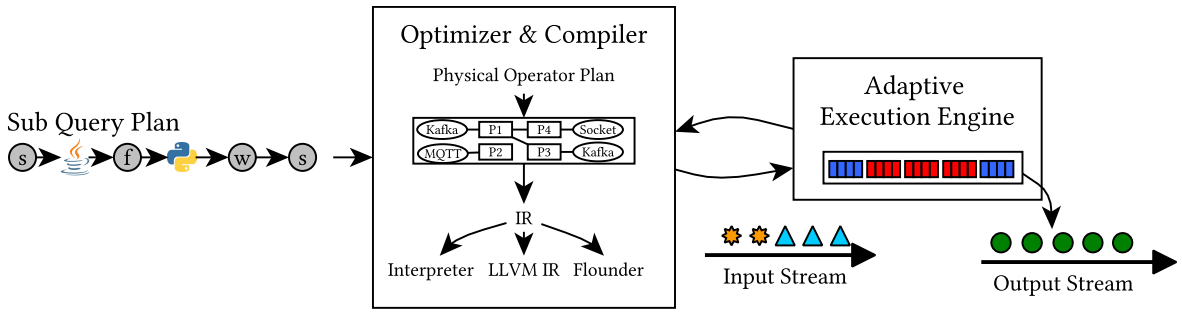


Figure 2.8: Overview of the query execution engines of NebulaStream.

2.3 Query Compilation in NebulaStream

Throughout this thesis, we rely on NebulaStream [289] as our target system and testbed of our contributions. In this section, we provide an overview of NebulaStream's execution engine architecture and highlight specific aspects of its query compilation approach.

NebulaStream is a general-purpose distributed data processing system. It targets a variety of data processing workloads and deployment environments with a primary focus on heterogeneous IoT environments and real-time data processing. To formulate queries, NebulaStream provides connectors to common data sources, e.g., MQTT, and Kafka, a Flink-like query language, and supports common data processing operators, e.g., `selections`, `map`, temporal `aggregations` and `joins`, as well as UDFs. Each NebulaStream deployment consists of a central *coordinator* and multiple potentially geo-distributed *worker* nodes. The coordinator receives user queries, performs global optimizations, and creates distributed sub-query plans. The worker nodes manage the execution of these sub-query plans, receive data from data sources, perform computations, and emit results to destinations and data sinks.

Consequently, NebulaStream's worker and its query execution engine are the primary focus of this thesis. Figure 2.8 illustrates a high-level overview of its execution engine architecture. Initially, the engine receives a logical sub-query plan that potentially combines different data processing operations, e.g., the example query performs a selection, two UDFs in Python and Java, and a temporal window aggregation. The execution of NebulaStream queries consists of three phases: *query segmentation*, *code generation*, *adaptive execution*. In the first phase, the engine translates the input query into physical operators. It provides traditional relation operators, stream operators, UDF-based operators, as well as system operators, e.g., to access the intermediate state. Based on the physical plan, the engine segments the plan in parallelizable pipelines and parameterized individual operators, e.g., to select materialization strategies. Thus, NebulaStream leverages a push-based execution strategy (see Section 2.2.1.1) in combination with task-based parallelization (see Section 2.2.1.3). In the second phase, the engine traverses the physical query plan and generates code fragments in its domain-specific IR (see Section 2.2.1.1). This decouples the implementation of operators from machine code generation and enables NebulaStream to leverage different compilation backends. For instance, NebulaStream uses bytecode interpretation [163] on low-powered embedded IoT devices, generates highly efficient code using LLVM on servers for continuous streaming queries, and uses Flounder [103] to support short-running batch queries. In the third phase, NebulaStream executes the generated code variant and continuously collects profiling information to track

specific data characteristics, e.g., selectivity or data distribution. If these data statistics change, it re-optimizes the query and deploys a new code variant.

This architecture enables NebulaStream to optimize query execution for diverse workloads and hardware environments. For long-running queries, the engine can reach high throughput **M1** while it also can reach low-latency **M2** for short-running queries. Additionally, its intermediate representation provides high expressiveness **M3** and decouples operators from query execution to increase ease of use **M4**. In the remainder of this thesis, we discuss the following aspects of NebulaStream's compilation-based engine in detail: Chapter 3 and 4 extend the traditional query compilation strategy for stream processing and UDF-based workloads. Based on these concepts, Chapter 5 proposes a novel framework for developing compilation-based execution engines, which is used as the foundation in NebulaStream.

3

Query Compilation for Stream Processing

Over the last decade, real-time data processing applications such as online fraud detection [235], real time monitoring [270], and online machine learning and inference [123, 193] has led to a wide adoption of stream processing application. These perform long-running queries over high-velocity data streams that involve unique stream processing operations, e.g., windowed aggregations and temporal joins. As such, the adoption of these application has lead to the development of a variety of stream processing systems (SPSs) [49].

3.1 Introduction

Open-source SPSs, such as Flink [50] or Spark [16, 286], as well as managed services, such as Google Dataflow [13] and Decodable [77], scale-out executions to enable nearly unlimited scalability. To handle high-velocity data streams, large- scale internet companies deploy these systems at a massive scale and spend millions of dollars per day for the infrastructure [93].

Even though these SPSs are widely adopted, recent work [287, 294] revealed that state-of-the-art systems do not fully exploit the hardware resources of individual compute nodes. The authors identified three main reasons for this. First, these systems cause many instruction cache misses because they use an interpretation-based processing model. Second, they cause many data cache misses because they rely on managed runtimes. Third, they utilize sub-optimal parallelization strategies on single nodes because they optimize for a scale-out environment. As a result, users are forced to scale out computation to reach a desired performance level, which leads to over-provisioning and a waste of compute resources. By eliminating these bottlenecks in hand-written implementations, Zeuch et al. [287] showed that a significant performance improvement is possible. However, to hand-code queries is impractical and cumbersome in practice. For databases, Neumann [202] introduced query compilation to achieve the performance of hand-written code for general query processing. However, no

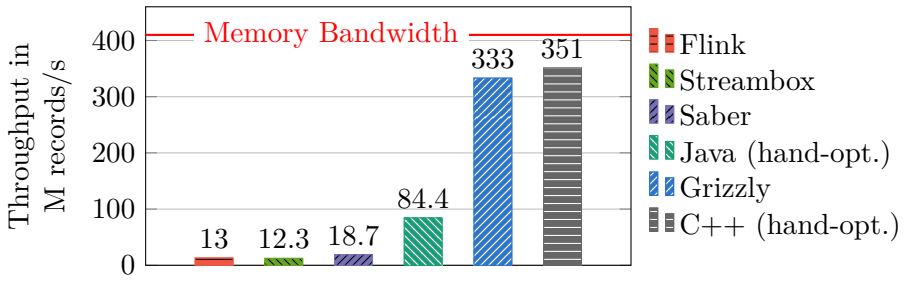


Figure 3.1: Throughput of state-of-the-art SPSs, hand-written implementations, and Grizzly on the Yahoo! Streaming Benchmark (8 Threads).

state-of-the-art SPE exploits query compilation to achieve similar performance improvements for streaming workloads.

In this chapter, we introduce Grizzly, the first adaptive, query compilation approach for stream processing on multi-core systems. Grizzly combines the generality and ease of use functionality of SPEs with the efficient hardware utilization of hand-written code. To reach this goal, we tackle three fundamental challenges: First, the semantics of stream processing are fundamentally different from relational algebra. Data streams are conceptually unbound and need to be discretized into finite windows. Windowing semantics are extremely diverse and cover different window types (e.g., tumbling and sliding windows), window measures (e.g., time-based and count-based windows), and window functions (e.g., aggregations). The cyclic control flow between these components makes it hard to apply state-of-the-art query compilation approaches directly. More specifically, Grizzly extends classic query compilation by supporting cyclic compile-time dependencies between the windowing components (assignment, trigger, and functions). Second, stream processing queries are inherently long-running, while the input stream constantly changes. As a consequence, the optimal plan and code efficiency changes over time too. To handle this, Grizzly establishes a feedback loop between code generation and execution. Our key idea is to use query compilation to inject low-overhead profiling code into the compiled query. The instrumented query collects profiling information, which we use to optimize the query at run time adaptively. Third, in contrast to relational operators, stream processing requires ordering between records. This introduces additional challenges for the concurrent processing of stateful operations (e.g., joins and aggregations). SPEs, such as Flink, apply key-by partitioning to mitigate this problem. In contrast, we apply task-based parallelism to utilize multi-core systems efficiently. To this end, Grizzly generates specialized code that ensures correctness and that takes the underlying hardware into account.

By tackling these challenges, Grizzly closes the gap between state-of-the-art SPEs and the performance of hand-written code. In Figure 3.1, we compare the performance of Grizzly to a scale-out SPE (Flink[50]), two scale-up SPEs (Saber [164], Streambox [196]), and two hand-optimized implementations of the Yahoo! Streaming Benchmark [67]. Only Grizzly and the hand-written C++ implementation fully utilize the available hardware. Grizzly outperforms state-of-the-art SPEs by up to an order of magnitude without losing generality. In summary, our contributions are as follows:

1. We present an adaptive query compilation approach for stream processing that generates efficient code.

2. We extend query compilation to support common window types, window measures, and window functions.
3. We introduce adaptive optimizations to react to changing data characteristics.
4. We utilize order-preserving task-based parallelization and introduce an efficient thread coordination protocol.
5. We demonstrate Grizzly's performance in comparison to state-of-the-art SPEs on diverse workloads.

The remainder of this chapter is structured as follows. First, we provide foundational background on stream processing and window semantics (see Section 3.2). Second, we introduce the architecture of Grizzly (see Section 3.3), its code generation approach (see Section 3.4), its parallelization technique (see Section 3.5), and its adaptive optimizations (see Section 3.6). Finally, we evaluate Grizzly (see Section 3.7) and discuss related work (see Section 3.8).

3.2 Stream Processing and Window Semantics

Processing contiguous, high-velocity, and unbounded data streams is a key requirement of Stream Processing Systems. These data streams are produced by sources like software and hardware sensors, transactions, or clicks. They push tuples to SPSs, where they arrive at high velocity in large volumes. Thus, storing the conceptually unbounded data streams is impossible. To this end, SPSs process and analyze data online, i.e., whenever tuples arrive.

Stream processing has been formally defined by multiple authors [41, 51, 164]. Following Carbone et al. [51] we define a data stream $\overline{s}$ as a sequence of records and denote, $s_i = \overline{s}(i)$ as the ith element in $\overline{s}$ and $\overline{s}([a,b]) = \overline{s}(R) = \{s_i | i \in R\}$ as a sub-stream of $\overline{s}$. To perform finite computation, e.g., aggregation, over unbounded streams, the window operator discretizes the data stream $\overline{s}$ into a sequence of potentially overlapping windows $w_i = \overline{s}([b_i, e_i])$. In the remainder of this section, we discuss the semantics of the window operator in detail.

3.2.1 Window Semantics

Windowing is a key aspect of modern SPSs and enables stateful operators, e.g., aggregations and joins, on unbounded data streams. Windows are characterized by a window type, a window measure, and a window function [268].

Window Types. The window type is formally defined by an assignment function $f_a(s_i) \rightarrow w_i$ that assigns a record s_i to a window w_i. Common window types are tumbling, sliding, and session windows [268]. Tumbling and sliding windows discretize a stream into windows of fixed length l. Additionally, sliding windows define a slide step l_s that declares how often new windows start. Thus, records are assigned to multiple concurrent overlaps sliding windows if $l_s < l$. Note that tumbling windows are a special case of sliding windows where the window size equals the slide size [275]. For instance, a sliding window could compute the average stock price over the last 5 minutes (size), updated every second (slide). In contrast, session windows are content-sensitive and end if no record is received for a time l_g (session gap) after a period of activity. If a new record arrives, a new session window starts and covers all records before the session timeout is passed.

Window Measures. The window measure defines the progress of windows. Common window measures are time and count [41]. We refer to windows using these measures as time-based and count-based windows, respectively. Time-based windows utilize a monotonic increasing timestamp ts and trigger as soon as the time passes the window end $ts > w_i.e$. In contrast, the length l of count-based windows corresponds to the number of assigned records. Thus, a count window ends if $i > w_i.e$. Note for keyed aggregations, time-based windows trigger for all keys at the same time, but the trigger decision of count-based windows has to be managed per key. For example, a content-based tumbling window with a size of n records ends as soon as the window contains n records.

Window-Functions. Window functions execute arbitrary computations on assigned records. For aggregation functions, we differentiate between decomposable and non-decomposable functions as proposed by Jesus et al. [148]. Decomposable aggregate functions (e.g., sum, avg) are computed incrementally; thus, only a partial aggregate has to be stored. In contrast, non-decomposable aggregation functions (e.g., holistic functions) require access to all records of a window. A function is decomposable if it is commutative and associative (e.g., sum, $count$, min, and max) or can be defined as a final aggregation function over a set of decomposable functions ($average = \frac{sum}{count}$).

3.3 Grizzly

In this section, we introduce Grizzly, our novel adaptive, compilation-based SPE. Grizzly's primary goal is to provide a high-level query interface for end-users while at the same time achieving the performance of hand-optimized code. In the remainder of this section, we discuss the major challenges of compilation-based SPEs (Section 3.3.1), present how Grizzly's core principles address them (Section 3.3.2) and explain Grizzly's execution model (Section 3.3.3).

3.3.1 Challenges for compilation-based SPEs

Similar to query compilation for data-at-rest, a compilation-based SPE, segments queries into multiple pipelines and fuses operators within pipelines. However, stream processing workloads introduces several unique challenges.

Challenge 1: Stream processing semantics. To the best of our knowledge, there is no SPE that is able to fuse stream processing queries involving windowing. The main challenges are three-fold. First, the window triggering depends on the window assignment and is order-sensitive. Second, the window function needs to be performed after the windowing, but defines the state that needs to be stored in windows. Third, triggering involves a final aggregation step (e.g., to compute the average). The cyclic control flow between these three tasks makes it hard to apply state-of-the-art query compilation techniques to an SPE because they assume only linear compile-time dependencies between operators.

Challenge 2: Order preserving semantics. In contrast to relational algebra, the outcome of stream processing operators depends on the order of records in the data stream. Thus, data-parallel execution requires coordination among processing threads before the next pipeline can process window results. A compilation-based SPE has to take this requirement into account during code generation. As a result, a compilation-based SPE has to adjust the

coordination among threads depending on the query to ensure correct processing results while enabling efficient processing.

Challenge 3: Changing data characteristics. Stream processing queries are deployed once and executed for a long time, while the input stream may change. In particular, they may face unpredictable changes in the data characteristics at runtime, e.g., a changing number of distinct values or a changing data distribution of keys. As a consequence, the efficiency of generated code may change over time. Thus, a compilation-based SPE has to re-evaluate the applied optimizations and, if required, generate new code during runtime.

3.3.2 Core Principles of Grizzly

Grizzly addresses the challenges introduced in Section 3.3.1, by applying query compilation, enabling task-based parallelization, and adaptively optimizing the generated code with regards to hardware and data characteristics.

Query Compilation. Grizzly introduces query compilation for stream processing and handles the complexity of windowing. Within pipelines, Grizzly fuses operations to compact code fragments and performs all operations of a pipeline in one single pass over a chunk of input records without invoking functions. Thus, data remains in CPU registers as long as possible without loading records repeatedly. To improve data locality in contrast to managed run-times, Grizzly avoids serialization and accesses all data via raw memory pointer. As a result, query compilation in Grizzly increases code and data locality significantly.

Order preserving task-based parallelization. To exploit multi-core CPUs efficiently, Grizzly executes pipelines concurrently in a task-based fashion on a global state. This eliminates the overhead of data pre-partitioning and state merging. However, it requires coordination between threads to fulfill the order requirements of stream processing. Grizzly addresses these by introducing a light-weight, lock-free window-processing approach based on atomics.

Adaptive optimizations. Grizzly introduces a feedback loop between code generation and query execution to exploit dynamic workload characteristics. Grizzly continuously monitors performance characteristics, detects changes, and generates new code variants. As a result, Grizzly performs speculative optimizations and assumptions about the incoming data. If an assumption is invalidated, Grizzly re-optimizes a code variant. To reduce the performance overhead, Grizzly combines light-weight but coarse-grained hardware performance counters with fine-grained code instrumentalization.

3.3.3 Compilation-based Query Execution

In Figure 3.2, we present the architecture of Grizzly's compilation-based query execution model, which consists of four phases. From the logical query plan ① to the continuous adaption to changing data characteristics ④.

3.3.3.1 Logical Query Plan.

In the first phase ①, Grizzly offers a high-level Flink-like API and translates each query to a logical query plan. This plan contains a chain of operators that consumes a stream with a static source schema. Grizzly supports traditional relational operators, e.g., selection and

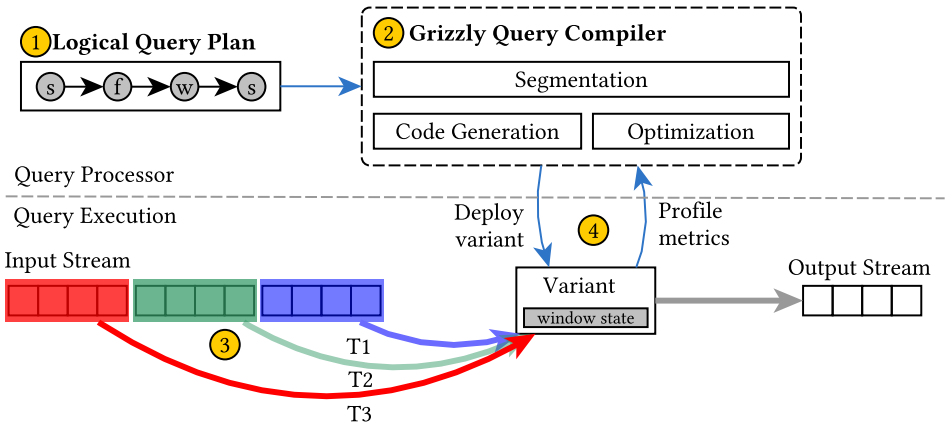


Figure 3.2: Overview of the query execution workflow in Grizzly.

map, and stream processing specific operators for windowing. Window definitions consist of a window type, a window measure, and a window function, as introduced in Section 3.2.1. Furthermore, Grizzly supports global windows that create one aggregate over the whole stream and keyed windows that created partitioned aggregations per key. Based on these operators, Grizzly supports common stream processing queries.

3.3.3.2 Query Compiler.

In the second phase ②, Grizzly segments the logical query plan into pipelines, performs optimizations, and generates code for each pipeline.

Segmentation. Query compilers for data-at-rest fuse operators until they reach a *pipeline-breaker*, which requires a full materialization of intermediate results (e.g., joins or aggregations). However, the unbounded nature of data streams prevents the full materialization of intermediate results. To this end, Grizzly separates pipelines at operators that require partial materialization, similar to soft-pipeline-breakers [287]. In particular, non-blocking operators (e.g., map or filter) are fused. In contrast, all blocking operations in stream processing are computed over windows (e.g., aggregations or joins) and terminate pipelines. Thus the support of windowed operations is crucial for a compilation-based SPE.

Optimization. After query segmentation, Grizzly optimizes the individual pipelines. To this end, Grizzly exploits static information, e.g., the hardware configuration, as well as dynamic data characteristics. To collect data characteristics, Grizzly introduces fine-grained instrumentation into the generated code. This enables Grizzly to derive assumptions about the workload, e.g., predicate selectivity and the distributions of field values. Based on these assumptions, Grizzly chooses particular physical operators.

Code Generation. In the last step, Grizzly translates each physical pipeline to C++ code and compiles it to an executable code variant. Note that all variants of the same pipeline are semantically equivalent, but execute different instructions and access different data structures. For code generation, Grizzly follows the produce/consume model and extends it with support for rich stream processing semantics. In particular, we consider code generation and operator fusion for the window operator.

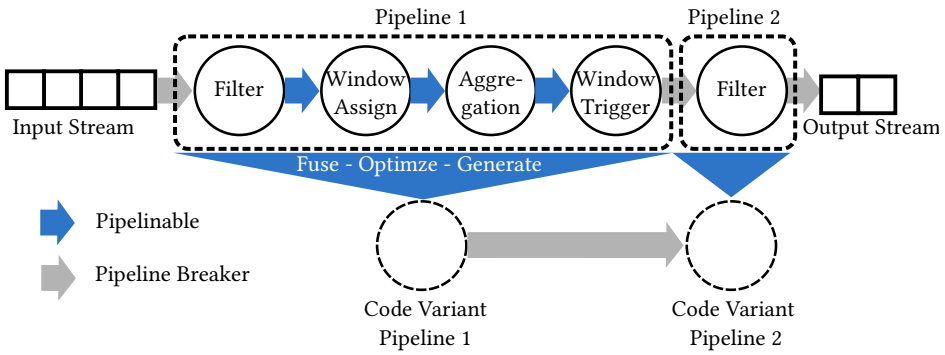


Figure 3.3: Query compilation for stream processing queries.

3.3.3.3 Execution

In the third phase ③, Grizzly executes the generated pipeline variant. Each variant defines an open and close function to manage the state of the variant. Depending on the physical operators, state is completely pre-allocated or dynamically allocated during execution. For the input stream, Grizzly exploits the fact that input records physically arrive in batches over the network and schedules each batch as a task for an individual thread to utilize multi-core CPUs. Thus, pipelines and their associated state are accessed concurrently by multiple threads. This introduces challenges for window processing, as all threads have to pass the window-end before one thread outputs the result. To this end, Grizzly introduces a lock-free data structure, such that multiple threads can concurrently process a window without starvation.

3.3.3.4 Profiling and Adaptive Optimization

In the final phase ④, Grizzly continuously collects profiling information and re-optimizes the query in two steps. During query execution, Grizzly collects hardware performance counters, e.g., the number of cache misses, to detect changing data characteristics. Hardware performance counters have a negligible performance impact [69, 292], but give a coarse-grained intuition about the evolution of data characteristics. If the collected counters indicate a change, Grizzly collects more fine-grained profiling information via code instrumentation. Based on this information, Grizzly re-optimizes the query and generates a new code variant.

3.4 Query Compilation in Grizzly

In this section, we detail Grizzly's query compilation approach and address the special challenges of stream processing. In particular, we focus on window aggregations as they are the primary operator requiring materialization and consequently break pipelines. Figure 3.3 illustrates how Grizzly segments an example query in two pipelines. Each pipeline begins with an arbitrary number of non-blocking pipeline operators (e.g., filter). Finally, each pipeline is terminated by the window operator, which Grizzly performs in three steps. First, the *window assigner* assigns input records, depending on the window type, to one or more corresponding windows. Second, the *window aggregator* updates the window aggregate. Third, the *window trigger* checks if an active window is complete and invokes the next pipeline to forward the window result. As a result, Grizzly supports a diverse set of window characteristics, which require specialized code

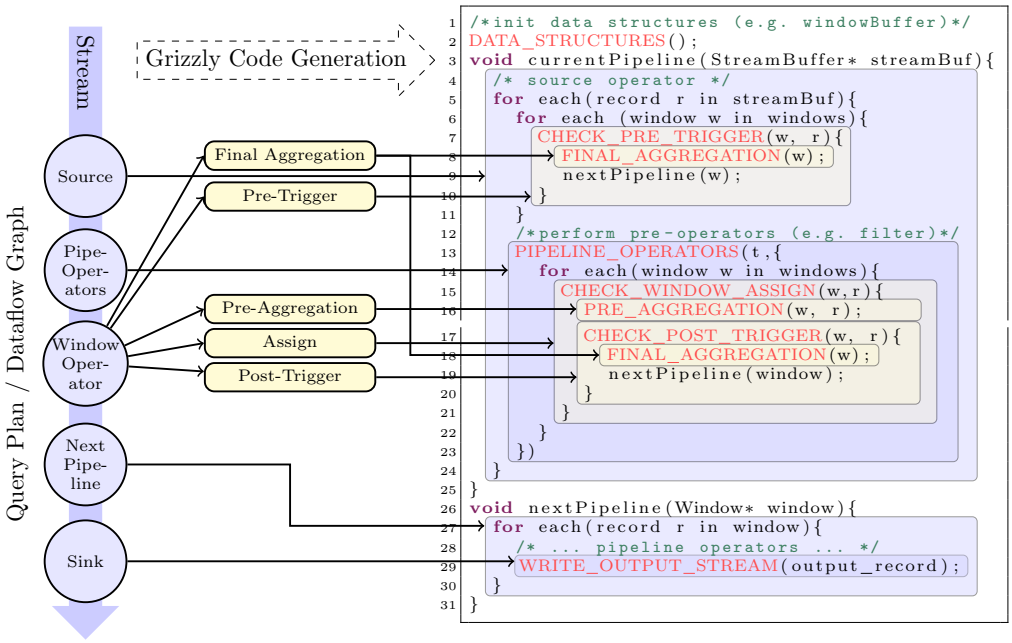


Figure 3.4: Mapping of a Logical Query Plan to a generic code template.

generation for all windowing aspects, e.g., assignment, aggregation, and trigger. In Figure 3.4, we present a mapping of a generic query plan to an abstract code template.

Note that we use the templates only for representation reasons, internally each physical operator produces C++ code depending on its particular properties. As shown, a stream processing query may consist of four types of operators: source, pipeline operators, window operators, and sinks. In the following, we first provide an overview of the operators that are supported by Grizzly (Section 3.4.1). After that, we describe the code generation for the windowing operator as the main building block in streaming queries (Section 3.4.2).

3.4.1 Operator Overview

In this section, we discuss the individual operators of Figure 3.4 and illustrate how Grizzly generates code for them.

Source Operator. The input stream arrives as a sequence of input buffers containing records. Each pipeline receives one input buffer at a time (Line 3) and the source operator iterates in a tight for loop over all records (Line 5). To avoid the deserialization of data from the input buffer, Grizzly casts the data from the raw buffer directly into complex event types. Then, the loop body executes all fused pipeline operators and ends with the window operator.

Pipeline-Operators. Pipeline-operators apply arbitrary non-blocking computation (e.g., filters, maps). Consequently, pipeline-operators could generate arbitrary output records per input record. Thus, all succeeding operations (e.g., window assignment and triggers) must be nested inside the pipeline-operators (Line 13).

Window Operator. Grizzly divides the window operator into three sub operators: assigner, aggregation, and trigger.

Assigner. During window assignment, Grizzly determines the target windows for the current record. The code iterates over all active windows and assigns the current record to its corresponding window(s) (Line 15).

Listing 2 Code generation template for the window assignment.

```
1 void generateAssignmentCode(WindowType type){
2   ts = getTimestamp();
3   if(type == TUMBLING_WINDOW){
4     PRE_AGGREGATE(w, r);
5     CHECK_POST_TRIGGER(w, r);
6   }else if(type == SLIDING_WINDOW){
7     if(ts >= w.begin && ts < w.end)
8       PRE_AGGREGATE(w, r);
9       CHECK_POST_TRIGGER(w, r);
10    }else if(type == SESSION_WINDOW){
11      PRE_AGGREGATE(w, r);
12      w.end = ts + type.timeout;
13      CHECK_POST_TRIGGER(w, r);
14    }
15  }
16 }
```

Listing 3 Code generation template for the window trigger.

```
1 void generatePreTrigger(Trigger t){
2   ts = getTimestamp();
3   if(w.end >= ts){
4     FINAL_AGGREGATION(w);
5     nextPipeline(w)
6     w.end = NEXT_WINDOW_END(w,ts);
7   }
8 }
9 void generatePostTrigger(Trigger t){
10  w.count++;
11  if(w.count == t.max_count){
12    FINAL_AGGREGATION(w);
13    nextPipeline(w);
14    w.count = 0;
15  }
16 }
```

Aggregation. After assigning a record to a window, Grizzly updates the window aggregate. Depending on the window function, Grizzly pre-aggregates records to minimize memory consumption (Line 16). After the window is triggered, Grizzly computes the final aggregate (Line 8 and Line 18).

Trigger. Depending on the window measure (count-based or time-based), it is required to perform the trigger check before (Line 7) or after the window assignment (Line 17).

Next-Pipeline. After triggering a window, the next pipeline starts processing window results (Line 26). The next pipeline can again contain arbitrary pipeline operators and ends with a window operator or a sink. As a result, Grizzly supports queries with multiple windows.

Sink Operator. The sink operator terminates a pipeline and writes records to an output stream (Line 29).

3.4.2 Window Operator

In this section, we discuss window operator-specific query compilation aspects. To this end, we present the code generation approach for window assignment (Section 3.4.2.1), window aggregation (Section 3.4.2.2), and the window trigger (Section 3.4.2.3). Finally, we discuss the handling of window joins (Section 3.4.2.4).

3.4.2.1 Window Assignment

The window assigner maps incoming records to windows. To this end, Grizzly keeps track of active windows and generates specialized code depending on the window type, illustrated in Listing 2. To keep track of active windows, Grizzly stores metadata for each window (e.g., start and end timestamps) in a compact array. During window assignment, Grizzly checks all windows and assigns the record to a window aggregate if the assignment condition is true. Depending on the window type, Grizzly generates different code. For tumbling and session windows, each record belongs to exactly one window (Line 3 and Line 10). In contrast, for sliding windows the generated code iterates over all active windows (Line 6) and selects matching windows based on the current timestamp (`ts`) (Line 7). Furthermore, session windows expand with each assigned record and Grizzly shifts the window end if a record is assigned (Line 12).

3.4.2.2 Window Aggregation

After assigning records to windows, Grizzly adds the record to the window aggregate. Grizzly differentiates between decomposable (e.g., *sum*, *avg*) and non-decomposable (e.g., *median*) aggregation functions as introduced in Section 3.2.1. For non-decomposable aggregation functions, Grizzly stores all assigned records in a separate window buffer and computes the final aggregation after the window is triggered. For decomposable aggregation functions, Grizzly computes the window aggregate incrementally. Thus, Grizzly only stores a partial aggregate instead of all assigned records, which reduces memory requirements. Furthermore, primitive partial aggregates can be updated much more efficiently using atomic operations. For keyed (grouped) aggregations, Grizzly maintains a partial aggregate per key.

3.4.2.3 Window Trigger

The window trigger finalizes windows and passes them to the next pipeline. Grizzly evaluates trigger conditions before the processing of a record (pre-triggers) or after the window aggregation (post-triggers). We illustrate the code generation algorithm in Listing 3.

Pre-Trigger. The pre-trigger checks active windows before processing the current record. This is necessary to support time-based window measures. Time triggers only depend on the progress of time for the trigger decision and are independent of the individual record. Thus, time-based triggers replace the CHECK_PRE_TRIGGER macro in the generic code template (see Figure 3.4, Line 7). For time-based triggers, the generated code compares the current timestamp `ts` to the end time of each active window (Line 3). If the window end timestamp is passed, the window is triggered. In this case, Grizzly computes the final window aggregate (Line 4) and calls the next pipeline to process the window result (Line 5). Finally, Grizzly clears the window state, calculates a new window end timestamp, and updates the window metadata (Line 6). Note that an additional trigger is necessary if the arrival rate of new records is too slow to guarantee a constant evaluation of the trigger function.

Post-Trigger. The post-trigger is executed after assigning a record to a window. It replaces the CHECK_POST_TRIGGER macro in the generic code template (see. Figure 3.4, Line 17). The post-trigger only evaluates the assigned window instead of all active windows. Post-triggers are necessary to support count-based windows, which directly trigger a window after the last record is assigned. In contrast, a count trigger maintains a counter to keep track of the number of assigned records to each active window (Line 10). If the number of items has reached the maximal window count (Line 11), the trigger calculates the final aggregate (Line 12) and invokes the next pipeline (Line 13). Finally, Grizzly clears the window state and sets the window count to zero (Line 14).

3.4.2.4 Windowed Join

Grizzly supports windowed equal joins following the semantics of Flink [50]. For each input stream, Grizzly generates one code pipeline that maintains an intermediate join table. Grizzly reuses the window trigger code to discard the intermediate state as soon as the window ends. During execution, each pipeline concurrently assigns records to its local join table and probes

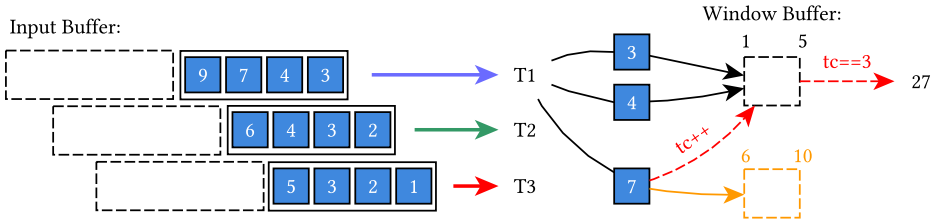


Figure 3.5: Example for Lock-Free-Window Trigger.

the record to the join table of the other join side. Consequently, the stream join is fully pipelined and non-blocking.

3.5 Parallelization

To utilize modern multi-core processors efficiently, Grizzly applies data-centric parallelization. This paradigm is reflected in both the classic exchange operator [116] and morsel-based operators [176]. Grizzly extends these ideas and introduces light-weight coordination primitives to address the unique ordering requirements of stream processing. During runtime, Grizzly creates tasks for each incoming buffer and its target processing pipeline. Worker threads execute the pipeline and operate on a shared global state, e.g., for window aggregations. This approach eliminates the data shuffling step of systems like Flink and provides robustness for skewed key distributions and heavy hitters. In the remainder of this section, we present how Grizzly coordinates window processing to address the order semantics of stream processing (Section 3.5.1) and how it specializes generated code with regards to NUMA hardware (Section 3.5.2).

3.5.1 Lock-Free Window Processing

In general, a dynamic, task-based parallelization can lead to wrong processing results for streaming queries. Thus, Grizzly has to prevent that windows are passed to the next pipeline, while other threads still assign records to them. A naïve approach would introduce a barrier at window ends to synchronize all processing threads. However, this limit performance due to the introduced waiting time. To overcome this limitation, we introduce a lock-free window processing technique that allows threads to process different windows concurrently. In particular, Grizzly maintains multiple window aggregates in a ring buffer (depending on the window type), similar to the technique proposed by Zeuch et al. [287]. Furthermore, each thread maintains a pointer to its current window aggregate and the value of the next window end. This technique enables Grizzly to support important properties. First, every thread can decide independently to which window it assigns incoming records. Second, only the last thread that modifies a window creates the final window aggregate and invokes the next pipeline.

Figure 3.5 illustrates an example of Grizzly's lock-free window implementation for time-based windows. Each thread processes its input buffer and checks per record if the window should trigger. If the window end is reached (at record ⑦ in Figure 3.5), the thread triggers the window *locally*. To this end, the thread calculates the next window end and shifts its current window pointer to the next position in the window-buffer. After that, the thread

Figure 3.6: NUMA-aware window computation in Grizzly.

will assign all succeeding records to the next window aggregate. In addition, each thread increments atomically a global trigger counter `tc`. If tc is equal to the degree of parallelism (tc==3 in Figure 3.5) it is guaranteed that all threads have triggered the window locally, and no thread will modify the window anymore. In this case, Grizzly creates the final window aggregate and invokes the next pipeline.

3.5.2 NUMA-aware Stream Processing

Research in the area of multi-core query execution shows that its crucial to take NUMA effects into account to enable scalability across multiple CPU sockets [158, 176]. Especially data accesses across NUMA regions reduce bandwidth by up to 2x [182]. In Grizzly, we minimize the inter-NUMA node communication and specialize the code generation to the underlying NUMA configuration. Figure 3.6, illustrates the NUMA-aware execution of a windowed stream processing query in Grizzly. During query compilation, Grizzly detects the NUMA configuration and deploys a two-phase strategy for window aggregations. In the first phase, processing threads pre-aggregate values into a hash map inside the local NUMA region. In the second phase, Grizzly merges the aggregates of the local states at the window end. During execution, Grizzly pins all processing threads to a specific NUMA region and only process local input buffers. Overall, this design reduces cross-NUMA communication to a minimum and enables efficient sharing inside one socket.

3.6 Adaptive Query Optimization

Research in the area of adaptive and progressive optimization demonstrates that the reaction to changing data characteristics improves performance significantly [21]. This specifically affects streaming queries, which are commonly deployed once and run virtually forever. In Grizzly, we detect and react to changing data characteristics at runtime and perform adaptive optimizations using JIT compilation. In the remainder of this section, we detail Grizzly's adaptive query compilation approach (Section 3.6.1) and present three optimizations that exploit specific data characteristics (Section 3.6.2).

3.6.1 Adaptive Query Compilation

Grizzly follows an explore/exploit approach to enable adaptive optimizations. At run time, Grizzly continuously performs optimization and deoptimization [96, 139]. Depending on assumptions about the workload (e.g., data- or hardware-characteristics), Grizzly generates specialized code variants. If assumptions become invalid, Grizzly deoptimizes and migrates back to a generic code variant. In the remainder of this section, we detail the individual steps of this process.

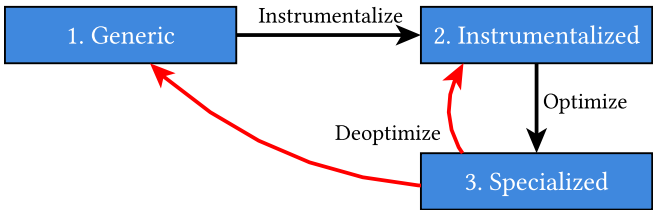


Figure 3.7: Execution Stages of Grizzly's adaptive compilation strategy.

3.6.1.1 Execution Stages

The following execution stages reflect Grizzlies adaptive compilation process, illustrated in Figure 3.7.

First Stage: Generic Execution. In the first stage, Grizzly executes a generic code variant and performs static optimization. For instance, Grizzly utilizes knowledge about the data schema to optimize comparisons to constant values.

Second Stage: Instrumented Execution. In the second stage, Grizzly introduces code instrumentation to collect fine-grained data-characteristics. Thus, each operator can generate arbitrary profiling instructions to track statistics (e.g., predicate selectivity or the domain of a value). To reduce overhead, Grizzly applies sampling by executing profiling code only with a subset of threads and on a subset of records.

Third Stage: Optimized Execution. In the third stage, Grizzly utilizes the profiling information to make assumptions about the underlying data characteristics. Based on this, Grizzly performs speculative optimizations and specializes code as well as data structures.

3.6.1.2 Deoptimization

Deoptimization migrates from the optimized code variant back to the generic one. The causes of this are two-fold. First, during execution, Grizzly detects that an assumption is violated. For instance, if a key exceeds the assumed range (*assuming* $x < 5$ *but actualy* $x = 10$). In this case, the current processing thread continuous with the generic code variant. Second, Grizzly continuously monitors hardware performance counters to identify changes in data-characteristics, e.g., number of cache misses. If Grizzly detects a change, it schedules the deoptimization of the current code variant. If the frequency of deoptimizations is low, Grizzly directly migrates to stage two.

3.6.1.3 Variant Migration

For the migration between code variants, Grizzly ensures correct query results while minimizing processing overhead. To this end, Grizzly lazily invalidates code variants such that multiple threads can operate on different variants concurrently. All processing threads determine the switch individually and switch to the next variant after the current task. If all threads have discarded the old variant, Grizzly triggers state migration. In the case of windows, this requires the merging of a specialized state representation with the generic representation of the same state. Furthermore, to ensure correctness, Grizzly triggers no windows before the migration is completed.

3.6.2 Adaptive Optimizations

In the following, we discuss three examples of adaptive optimization implemented in Grizzly. Beyond this, Grizzly's adaptive optimization approach is able to detect and react to a wide range of different characteristics (e.g., ingestion rate, value distribution, selectivity), and to perform a wide range of optimizations (e.g., operator re-order, algorithm selection, data-structure specialization).

3.6.2.1 Exploiting Predicate Selectivity

Optimizing selection operators has been studied extensively in the database [47, 48, 81, 92, 240, 280, 291] as well as the compiler community [15, 20]. In Grizzly, we utilize profiling information to determine the optimal order of selection predicates inside a query plan. In particular, conjunctions over multiple predicates benefit if the most selective predicate is evaluated first, as the CPU can skip the evaluation of all other branches. Additionally, predicates with a selectivity of around 50% cause miss-prediction and introduce a high-performance overhead. During *instrumentalization*, Grizzly generates one counter per predicate to measure the individual selectivity. In comparison to measuring the combined operator selectivity with performance counters [292], this allows to choose the optimal predicate order directly. During *optimized execution*, Grizzly executes the optimized code variant and monitors the number of mispredictions for taken and not taken branches by applying the cost model of Zeuch et al. [291]. An increasing number of mispredictions indicates that the selectivity of a predicate changed and that the current predicate order becomes inefficient. Thus, Grizzly initiates a new profiling phase to reoptimize the predicate order.

3.6.2.2 Exploiting Value Ranges

In the general case, Grizzly maintains window aggregates in an Intel TBB concurrent hash-map [144]. This hash map accepts any data type for keys and values and grows dynamically with the number of keys. As a result, Grizzly supports any number of input keys as long as the hash map fits into memory. However, this flexibility introduces a substantial overhead [187]. To mitigate this overhead, Grizzly speculates on the value range. During instrumentalization, Grizzly injects code to identify the maximal and minimal key value that is inserted into the map. During *optimized execution*, Grizzly replaces the dynamic hash-map with a static memory buffer, which only stores window aggregates. This prevents hash collisions and eliminates overhead for resizing the state. To prevent out-of-bound accesses, Grizzly de-optimizes the code variant if a key lies outside of the assumed value range. This additional check introduces a negligible overhead as the condition is false as long the assumption is valid. Thus, the CPU branch predictor can predict the branch always correctly.

3.6.2.3 Exploiting Value Distributions

The efficiency of window aggregations highly depends on the hash-map implementation and the key distribution in the workload [68]. A global shared hash map is beneficial for uniformly distributed keys, as concurrent accesses to the same key are less frequent. In contrast, skewed workloads with heavy hitters benefit from an independent hash map per thread. This eliminates

concurrent accesses and synchronization overhead but requires merging and reduces memory efficiency as aggregates are stored multiple times. Grizzly adaptively chooses between both strategies depending on the data characteristics. During instrumentalization, Grizzly creates a histogram over the keyspace to monitors the distribution. If Grizzly can assume that the majority of accesses could hit at least the L3 cache, Grizzly uses the independent hash map. During *optimized execution*, Grizzly monitors the performance counters of the cache coherence protocol to detect if the selected strategy is still appropriate. For instance, an increasing number of exclusive accesses to a cache line that another thread has in exclusive access indicates that the uniform distribution shifts to a more skewed distribution.

3.7 Evaluation

In this section, we experimentally evaluate Grizzly. In Section 3.7.1, we introduce our experimental setup. After that, we conduct four sets of experiments. First, we evaluate the throughput and latency of Grizzly and state-of-the-art SPEs for different workloads (Section 3.7.2). Second, we highlight the throughput impact of different workload characteristics (Section 3.7.3). Third, we showcase the advantages of Grizzly's adaptive optimizations (Section 3.7.4). Finally, we analyze resource utilization and system efficiency to reveal the reasons why Grizzly's utilize modern hardware more efficiently compared to state-of-the-art SPEs (Section 3.7.5).

3.7.1 Experimental Setup

In the following section, we present the hardware and software configurations (Section 3.7.1.1) and the workloads of our experiments (Section 3.7.1.2).

3.7.1.1 Hardware and Software

We execute experiments on two machines: a commodity, single-socket server (Server A) and a high-end, multi-socket server (Server B) (to isolate the effects of NUMA). Server A has one Intel Core i7-6700K processor with four physical cores (in total 8 logical cores) and contains 32GB main memory. Server B has two Intel Xeon 6126 with 12 physical cores each (in total 48 logical cores) and contains 1.48TB main memory. Both CPUs have a dedicated 32 KB L1 cache for data and instructions per core. Additionally, Server A has 256 KB L2 cache per core and 8 MB L3 cache per CPU, and Server B has 1MB L2 cache per core and 19.25 MB L3 cache per CPU. If not stated otherwise, we execute all experiments on Server A using all logical cores.

The C++ implementations are compiled with GCC 6.5 and O3 optimization, as well as the mtune flags to produce specific code for the underlying CPU. We measure hardware performance counters using PAPI [264] version 5.5.1. The Java implementations run on the HotSpot VM in version 1.8.0 201. We use Apache Flink [50] in version 1.8.0 as a representative scale-out SPE and disable fault-tolerance mechanisms to minimize overhead. As representative scale-up SPEs, we use Streambox [196] (C++ based) and Saber [164] (JVM-based). In the following evaluation, we examine two versions of Grizzly. Grizzly refers to a version that does not exploit any knowledge about the data characteristics and thus applies no adaptive,

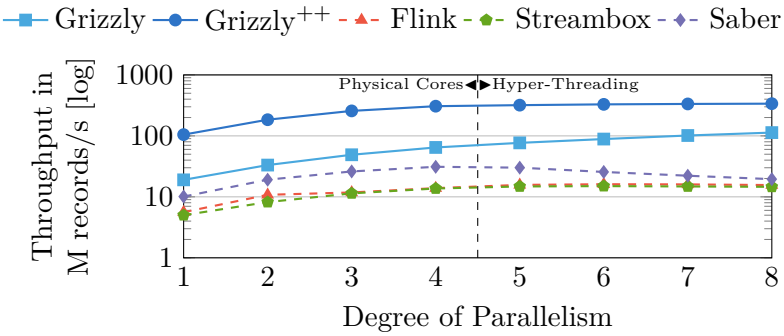


Figure 3.8: Scaling parallelism (1-8 threads) on the YSB query.

data-driven optimizations. Grizzly^{++} refers to a version that is aware of data characteristics and thus applies the adaptive, data-driven optimizations from Section 3.6.2.

3.7.1.2 Workload

If not stated other, we base our experiments on variations of the Yahoo! Streaming Benchmark (YSB) [67] to simulate real-world stream processing workloads. We follow the YSB implementation of Grier et al. [118] and Saber [223], which processes all data directly inside the SPE to prevent the overhead of external systems such as Apache Kafka or Redis. The YSB query consists of two processing steps. First, the YSB query evaluates if the event type matches the string *view* (33% of the records qualify). Second, the YSB query aggregates the qualifying records by their campaign id into a processing-time tumbling window of 10 seconds. We ingest data with 10k distinct keys and process a SUM aggregation.

3.7.2 System Comparison

In this section, we study the system throughput under the impact of parallelism (Section 3.7.2.1 and Section 3.7.2.2), compare processing latencies (Section 3.7.2.3), evaluate queries from the Nexmark benchmark (Section 3.7.2.4), and discuss all findings (Section 3.7.2.5).

3.7.2.1 Scaling on a Single Socket

In this experiment, we evaluate the scalability of Flink, Streambox, Saber, and Grizzly on Server A. We execute the default YSB query and study the throughput for an increasing degree of parallelism.

Results. In Figure 3.8, we scale the execution of the YSB benchmark using different degrees of parallelism. Flink and Streambox scale up similar and achieve a throughput of up to 16M records/s. In contrast, Saber outperforms Flink and Streambox by 2.2x (31M records/s). Saber's throughput increases up to four cores. Beyond that, the throughput decreases due to hyper-threading. Hyper-threading (HT) introduces two logical cores for each physical core, which share caches, branch prediction units, and functional units [97]. HT is beneficial if multiple threads execute different types of work (e.g., computation and I/O accesses) [298]. Therefore, the results for Saber indicate that multiple threads compete for the same shared CPU resources, which limits the performance improvements of HT [97]. Note that the results are in line with numbers published by the original authors [222]. As shown, both versions of

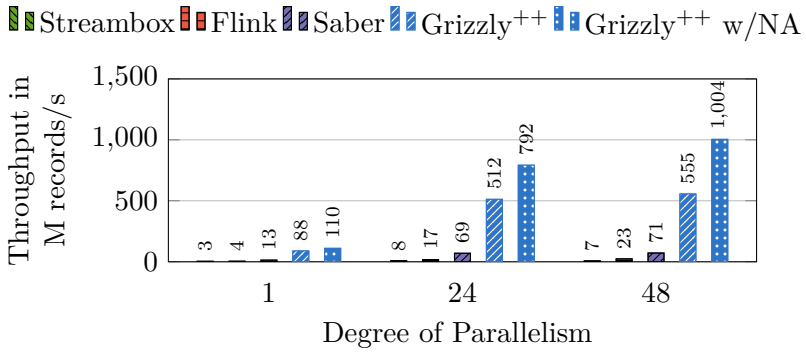


Figure 3.9: Scaling parallelism on a NUMA system (1-48 threads) using the YSB query.

Grizzly outperform all other SPEs. In particular, Grizzly achieves near-linear speedup and exploits HT efficiently. In contrast, by exploiting adaptive optimizations, Grizzly^{++} achieves the highest throughput, which is over an order of magnitude higher compared to Flink, Saber, and Streambox. Furthermore, Grizzly^{++} becomes memory bound for a degree of parallelism of four (all physical cores), and thus HT does not improve throughput significantly. Overall, without adaptive optimizations, Grizzly outperforms Saber by an average factor of 2.9 (min 1.7x, max 5.8x) and Flink/Streambox by a factor of 5.3 (min 3.7x, max 7.7x). With adaptive optimizations, Grizzly^{++} achieves an average speedup of 4.2x (min 2.9x, max 5.4x) over the generic Grizzly version (due to its more dense memory layout). As a result, Grizzly^{++} outperforms all evaluated SPEs on average by at least one order of magnitude (Saber 11.5x, Streambox 21.4x, Flink 21.5x).

3.7.2.2 NUMA Scaling

In this experiment, we evaluate the scalability of all SPEs on Server B. For Grizzly, we differentiate between a NUMA-aware version as outlined in Section 3.5.2 (Grizzly^{++} w/ NA) and a NUMA-unaware version (Grizzly^{++} w/o NA). We execute the YSB query and compare the throughput for parallelism of 1, 24, and 48.

Results. Figure 3.9 highlights the impact of NUMA for the individual systems. Overall, this experiment highlights the impact of NUMA-awareness. Already, for parallelism of one, Grizzly^{++} w/ NA leads to a speedup of 1.3x as it guarantees that all data is located on the same NUMA node as the processing thread. By increasing the degree of parallelism to 24, all systems improve throughput. In this case, Grizzly^{++} w/ NA results in a speedup of 1.5x compared to Grizzly^{++} w/o NA. If we further increase the degree of parallelism to 48, we observe that the throughput of all NUMA-unaware systems stagnates. In contrast, Grizzly^{++} w/ NA optimizations result in an additional speedup of 1.8x.

3.7.2.3 Latency

In this experiment, we examine the processing latency. First, we study the dependency between the buffer size and the latency for Grizzly. Additionally, we compare the latency of Grizzly, Saber, Streambox, and Flink. We define latency as the duration between the ingestion time of the last record that contributes to a window aggregate and the output of the aggregate of that window [153]. We execute the YSB on all systems with a parallelism of eight.

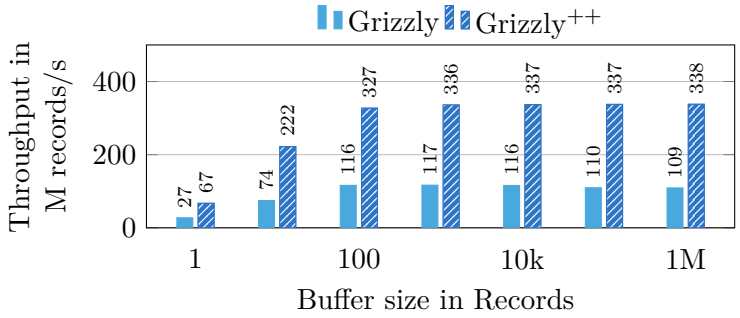


Figure 3.10: Scaling input buffer size.

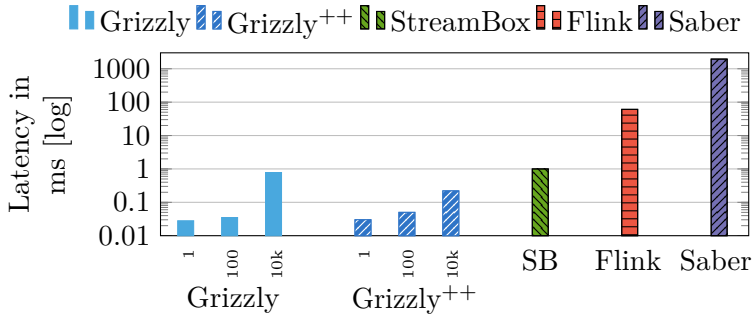


Figure 3.11: Processing Latency of Grizzly, StreamBox, Flink, and Saber.

Results. In Figure 3.10, we observe that both Grizzly versions reach peak performance for a buffer size larger than 100 records as the run time overhead becomes neglectable [290]. For the dependency between buffer size and latency, Figure 3.11 highlights two aspects. First, the buffer size has a high impact on the processing latency of Grizzly. For a buffer size of one, Grizzly achieves an average latency of 0.035ms (± 0.014ms) that increases up to 0.91ms (± 0.26ms) for a buffer size of 10k records. This characteristic is independent of the Grizzly version. Second, the code optimizations of Grizzly^{++} lead to lower latencies and smaller variances for large buffer sizes (avg. latency 0.22ms ± 0.15ms). The main reason for this is the higher complexity of the TBB hash-map in the default Grizzly version. Streambox is the only SPEs that is also able to reach average latencies in the range of 1ms (± 0.4ms). In contrast, Flink has on average a latency of 60ms (± 4ms) and Saber 1.9s (± 49ms). The higher latency of Saber is caused by its micro-batch processing model [164]. The micro-batching approach trades higher throughput for higher latency and is one of the reasons why Saber's throughput is higher compared to Flink and Streambox. Overall, both versions of Grizzly achieves up to an order of magnitude lower latencies and smaller latency variance than all other SPEs.

3.7.2.4 Nexmark Benchmark

In this set of experiments, we evaluate five queries of the Nexmark benchmark [271] on Grizzly^{++} and Flink. In particular, we use a tumbling window of 10s for Q7 and Q8, a sliding window of 10s with a slice of 1s for Q5, and a Sum aggregation for Q5 and Q7. In contrast, Q1 and Q2 are window-less.

Results. In Figure 3.12, we present the throughput of queries with different workloads on both systems. For the stateless Map (Q1) and Filter (Q2) queries, Grizzly^{++} outperforms

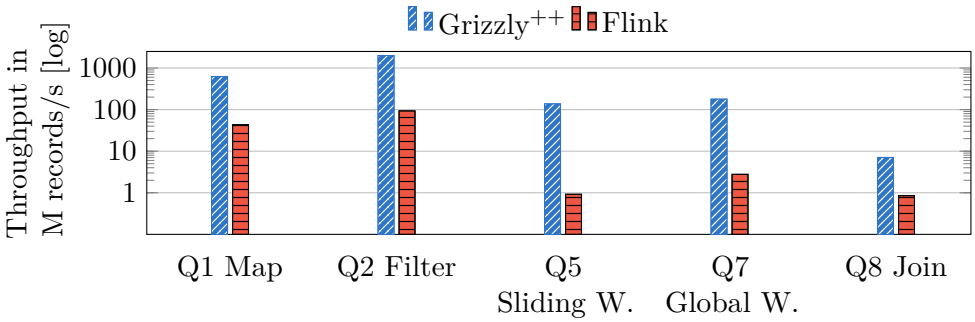


Figure 3.12: Comparison of queries from the Nexmark benchmark on Grizzly and Flink.

Flink by at least 14x. Flink and Grizzly perform these queries without any coordination between threads and process every input record only once. However, Grizzly^{++} benefits from eliminating any data serialization overhead. For the stateful queries Q5 and Q7, Grizzly^{++} outperforms Flink by at least a factor of 60x due to its more compact state representation, which improves cache locality. Additionally, Grizzly's task-based parallelization technique is beneficial for Q7. In contrast, Flink cannot parallelize the processing of global windows. The stream join of Q8 is highly resource-intensive, as both systems have to materialize the complete input data stream until the window triggers. Grizzly^{++} concurrently builds and probes the join tables across all processing threads, which introduce additional coordination overhead. However, Grizzly^{++} still outperforms Flink by at least a factor 8x on Q8. Overall, we observe that Grizzly++ provides similar throughput improvements among all Nexmark queries and outperforms Flink by at least 8x. As our selected set of queries covers basic building blocks of queries, we expect similar performance improvements for other streaming workloads.

3.7.2.5 Discussion

Across all experiments, we observed, that Grizzly outperforms all evaluated systems by up to one order of magnitude in throughput as well as latency on commodity hardware as well as high-end NUMA servers. The code specialization based on data characteristics (Grizzly^{++}) increases the throughput by up to 5.4x compared to the version without code specialization (Grizzly). Starting from small buffer sizes of 100 elements, Grizzly^{++} reaches peak throughput (337 million records/s) and achieves sub-millisecond latencies. Therefore, Grizzly mitigates the trade-off between latency and throughput. This experiment highlights two important aspects of stream processing on modern hardware. First, both versions of Grizzly exploit the cores of the CPU efficiently and code generation leads up to an order of magnitude performance improvement. Second, the code specializations of Grizzly^{++} induce an additional speed up and are crucial to fully utilize modern hardware efficiently. Furthermore, we highlight that Grizzly supports a wide range of workloads and reaches high performance on complex operators such as joins or aggregations.

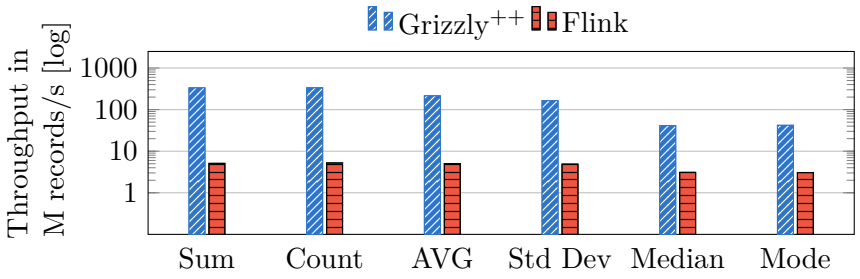


Figure 3.13: Throughput of decomposable (sum, count, avg, and std dev) and non-decomposable (median and mode) aggregation function on Grizzly and Flink.

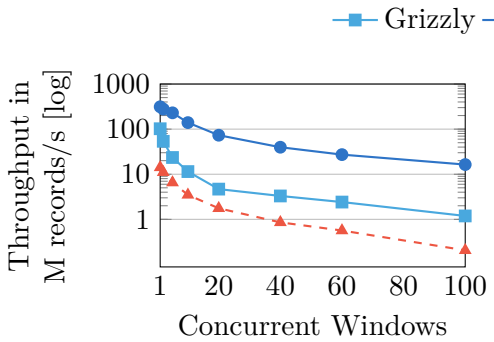


Figure 3.14: Throughput for concurrent time-based sliding windows.

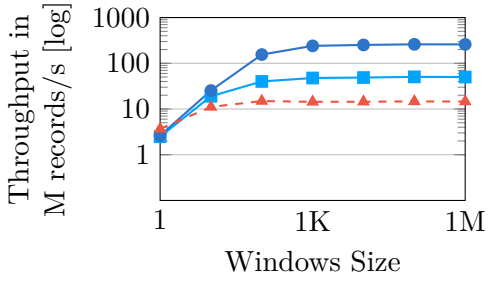


Figure 3.15: Throughput for tumbling count-based window.

3.7.3 Workload Characteristics

In the following set of experiments, we study the impact of particular workload characteristics on the throughput. To this end, we study the impact of state size (Section 3.7.3.4), the number of concurrent windows (Section 3.7.3.2), and count-based windows (Section 3.7.3.3).

3.7.3.1 Impact of Aggregation Type

In this experiment, we evaluate six window aggregation functions with a tumbling window of 10s on Grizzly^{++} and Flink. We evaluate four decomposable (i.e., *Sum*, *Count*, *AVG*, *StdDev*) and two non-decomposable aggregation functions (i.e., *Median*, *Mode*).

Results. The results in Figure 3.13 highlight the dependency between the complexity of the aggregation function and processing throughput. For decomposable aggregation functions, Flink reaches an average throughput of 5M records per second. In contrast, Grizzly^{++} outperforms Flink by a factor of up to 64x. Depending on the number of atomic state variables, Grizzly^{++}'s throughput varies up to a factor of 2x (e.g., SUM requires one atomic update, and Std Dev requires three updates per record). In the case of non-decomposable aggregation functions, the throughput of both systems decreases as they must materialize all records until the window ends. However, Grizzly^{++} is still able to outperform Flink by a factor of 13x. This is mainly due to its light-weight, in-memory state representation.

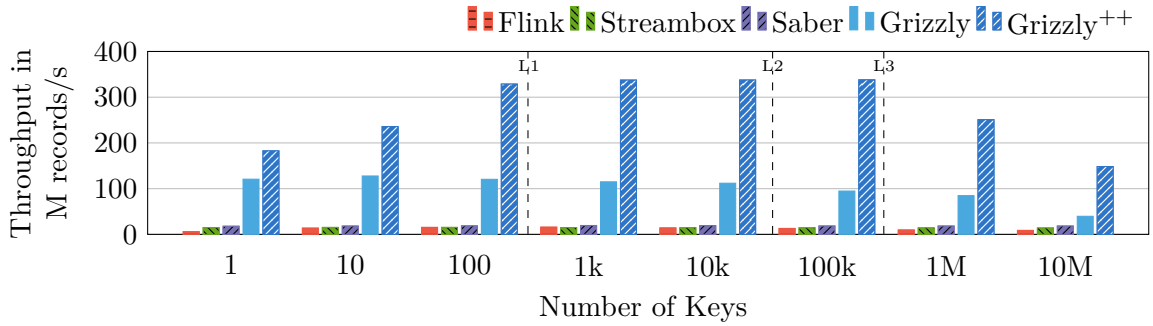


Figure 3.16: Throughput for scaling the state size.

3.7.3.2 Impact of Concurrent Windows

In this experiment, we study the throughput of over overlapping sliding windows. In particular, the efficient support of sliding windows is crucial as the ratio between size and slide could lead to high numbers of concurrent windows, e.g., a sliding window of one hour with a slice of one-minute results in 60 concurrent windows. We use the YSB query with a sliding window and scale the number of concurrent windows from 1 to 100.

Results. Figure 3.14 shows that the throughput of Flink and Grizzly is highly dependent on the number of concurrent windows. The overhead of concurrent sliding windows was demonstrated in previous work [261, 267, 268]. Both Flink and Grizzly use buckets to maintain window aggregates. Thus, both systems have to assign each record to multiple windows. As a result, the performance decreases with an increasing number of concurrent windows. However, Grizzly outperforms Flink on average by a factor of 4.2x (Grizzly) and 44x (Grizzly^{++}). Grizzly^{++} achieves a higher throughput as it represents state in a dense fixed-size array. This simplifies data access and reduces cache misses also in the case of concurrent windows.

3.7.3.3 Impact of Window Measure

In the previous experiments, we studied time-based windows. In contrast to time-based windows, the triggering logic of count-based windows is fundamentally different and more complex as it requires updating a global counter after each record assignment. In the following experiment, we study the impact of the size of a count window on the throughput. We execute the YSB query with a count-based window and vary the window size, which directly determines the window trigger frequency.

Results. Figure 3.15 reveals that the trigger overhead dominates the throughput for small window sizes (1-100 records) across all SPEs. Starting from a window size of 1k records, the overhead gets negligible, and the throughput becomes independent of the window size. For windows larger than 1k records, Grizzly outperforms Flink by a factor of 3.4x (generic Grizzly) and 17.7x (Grizzly^{++}).

In comparison to time-based windows, count-based windows reduce the throughput by a factor of two. This is mainly due to the more complex window trigger logic required for count-based windows (see Section 3.4.2.3). In particular, Grizzly maintains a counter per key and window, which has to be incremented atomically for each assigned value.

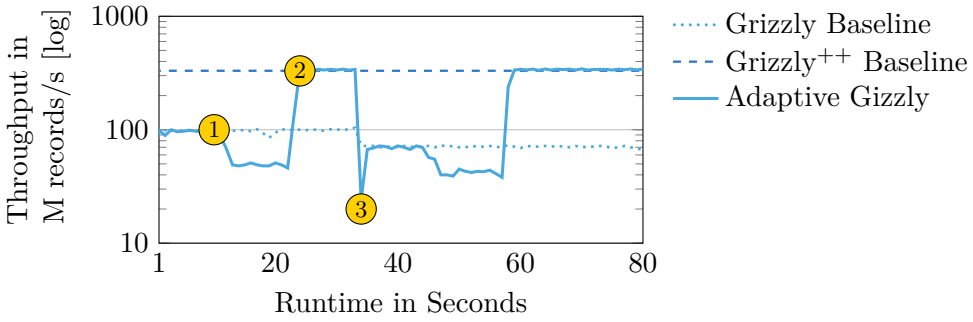


Figure 3.17: Changing value range.

3.7.3.4 Impact of State Size

In this experiment, we scale the state size by adjusting the number of distinct keys (8byte per campaign id) in the input data stream of the YSB query. In particular, we execute the default YSB query and scale the number of distinct keys from 1 to 10 million. Because the YSB query aggregates by key, the number of keys directly impacts the intermediate state size.

Results. Figure 3.16 highlights the dependency between throughput and state size (number of keys) across the examined systems. Streambox and Saber achieve, on average, a throughput of 15M and 19M record/s, respectively. Both systems outperform Flink that reaches the lowest average throughput with 11M records/s. If the stream only consists of one key, Flink's throughput decreases to 6M record/s, which is equal to its single-thread performance (see evaluation in Section 3.7.2.1). This demonstrates the disadvantage of key-partitioning based parallelization as only one thread performs computations per distinct key. In contrast, Grizzly outperforms all other SPEs for all state sizes.

In general, increasing key ranges of generated records induces only a small impact on throughput for Flink, Saber, and Streambox. If the intermediate state exceeds the L3 Cache (more then 130k keys), the throughput slightly decreases (e.g., for Flink 0.6x). This result indicates that all three SPEs do not exploit modern hardware, in particular, CPU caches, efficiently. In comparison to the best performing SPE (Saber), Grizzly reaches an average speedup of 5.9x (min 4.4x, max 7.0x) and Grizzly^{++} reaches an average speedup of 15.3x (min 10.2x, max 18.4x). For small state sizes (1-100 keys), Grizzly^{++} induces a high overhead. This overhead is mainly caused by concurrent accesses on a small number of keys that result in a significant synchronization overhead. Between 100 and 100k keys, Grizzly^{++} reaches peak performance (338M records/s). For more than 100k keys, the performance of both Grizzly versions decreases as the state size exceeds the L3 Cache.

3.7.3.5 Discussion.

In these experiments, Grizzly outperformed Streambox, Saber, and Flink across all tested workload configurations. Depending on the query workload, the performance improvements differ. In particular, the aggregation type, the window type, and the number of concurrent windows impact performance significantly. However, we proved that the adaptive optimizations of Grizzly^{++} exploit the cache hierarchy and the capabilities of modern hardware most efficiently.

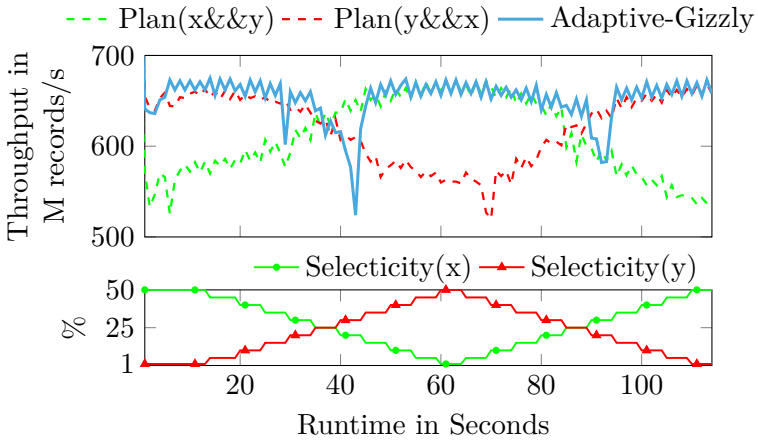


Figure 3.18: Changing predicate selectivity.

3.7.4 Adaptive Optimizations

In the following set of experiments, we evaluate Grizzly's adaptive optimization techniques (Section 3.6.2). To this end, we investigate the different data characteristics and their impact.

3.7.4.1 Compilation Stages

In this experiment, we study the performance impact of Grizzly's three compilation stages. We execute the YSB query and configure the duration of each compilation stage to 10 seconds. After 30 seconds, the number of distinct keys increases by 10x.

Results. Figure 3.17 illustrates the system throughput of Grizzly over time. At the beginning, Grizzly deploys the generic code variant and reaches a throughput of 100M records/s. At ①, Grizzly migrates to the instrumented code variant. The profiling instructions introduce an overhead of 50% such that the throughput decreases to 50M records/s. At ②, Grizzly utilizes the collected profiling information to deploy an optimized code variant. This results in a speedup of 3.3x over the generic baseline (330M records/s). At ③, the number of distinct keys increases and Grizzly de-optimizes the pipeline variant as discussed in Section 3.6.1. After deoptimization, the throughput drops shortly to 24M records/s, before a new optimization circle starts.

3.7.4.2 Selectivity Profiling

In this experiment, we study the performance impact of optimizing predicate reordering in queries containing selections (see Section 3.6.2.1). To this end, we introduce five greater equal predicates into the YSB query such that 120 different predicate orders are possible. During execution, we vary the selectivity of two predicates (x and y). All other predicates have a fixed selectivity of 50%.

Results. Figure 3.18 compares the throughput of Grizzly with two specific plans, which either first evaluate the x or y predicate. At the beginning, predicate y is very selective. Thus it is more efficient to evaluate y first. Starting from second 40, the predicate x becomes more selective then y. Thus, it evaluating x first reduces the number of branches. Grizzly detects the crossing point, and changes the operator order to evaluate x first. As a result, this adaptive optimization leads to a throughput difference of up to 150M records/s.

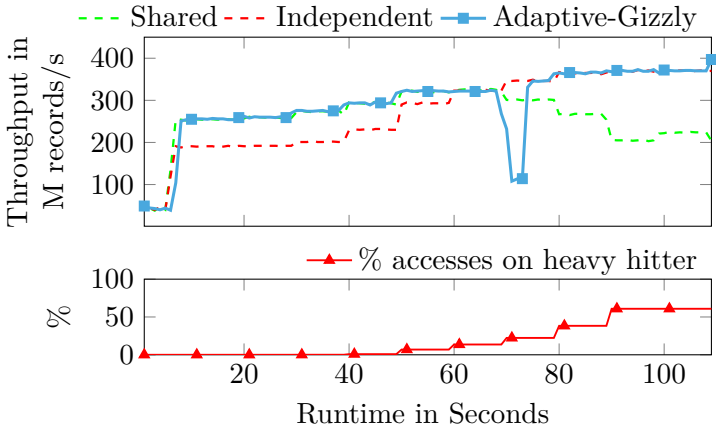


Figure 3.19: Changing key distribution.

3.7.4.3 Heavy-Hitter-Profiling

In this experiment, we study the impact of Grizzly's adaptive optimization for detecting the distribution of keys in window aggregations (see Section 3.6.2.3). We execute the standard YSB query with 1M distinct keys. Over time, we shift the distribution of keys, starting with a nearly uniform distribution towards a scenario where 60% of records access the same key.

Results. After the initial profiling phase, Grizzly detects that the keyspace is nearly uniformly distributed and chooses a shared hash-map. After 60 seconds runtime, the performance of the shared hash-map significantly decreases, as more than 10% of all records access the same key. Grizzly detects the increasing cache contention with performance counters and triggers re-optimizes by migrating to an independent hash-map. For highly skewed distributions, the independent hash-map achieves a speedup of up to 2x.

3.7.4.4 Discussion

The experiments showed that Grizzly is able to detect and exploit changing data-characteristics adaptively at runtime. Depending on the scenario, an optimized code variant can result in a performance gain of up to 3x. Thus, it is important to limit the execution time of unoptimized pipeline variants (e.g., by profiling only small buffers. Leis et al. already showed that 10k records are enough to identify join orders [179]).

3.7.5 Analysis of Resource Utilization

In this section, we evaluate the resource utilization of Grizzly, Streambox, Flink, and Saber. The resource utilization enables us to explain the different performance characteristics observed in previous experiments. In Table 3.1, we show performance counters for the default YSB query. These results reflect the pure execution workload per record without any preprocessing. We divide the counters into three blocks: Control Flow, Data Locality, and Code Locality.

Control Flow. In the first block, Table 3.1 shows the number of executed branches and branch mispredictions per record. These counters are essential to analyze the control flow of the SPEs. Across all SPEs, Streambox, and Flink introduce the highest number of branches and branch mispredictions. For Flink, data serialization and object allocation cause many dynamic branches and branch mispredictions, which was already shown by Zeuch et al. [287]. Saber

Table 3.1: Resource utilization per record on YSB query across different SPEs.

	Grizzly	**Grizzly^{++}**	**Streambox**	**Saber**	**Flink**
Branches/rec	18.2	7	706	184	701
Branch Mispred./rec	0.78	0.38	5.5	0.25	2.23
L1-D Misses/rec	3.5	1.39	50.6	11.9	23.8
L2-D Misses/rec	11.2	3.59	132	22.6	43.8
LLC Misses/rec	1.8	1.92	46.6	10.9	18.7
TLB-D Misses/rec	0.01934	0.01234	6.7	0.20	0.45
Instructions/rec	139.4	41.6	3440	1157	4162
L1-I Misses/rec	0.00026	0.00011	13.8	1.1	14.4
L2-I Misses/rec	0.00023	0.00010	2.7	0.26	1.2
TLB-I Misses/rec	0.00012	0.00006	0.091	0.017	0.081

achieves the fewest branch mispredictions but executes up-to 26x more branches compared to Grizzly^{++}.

The main reason for the high number of branches is Saber's micro-batch processing model, which performs many prediction-friendly branches by looping over data in batches. Overall, both versions of Grizzly introduce very few branches and branch mispredictions. Finally, we show that the adaptive optimizations in Grizzly^{++} reduce branches and branch mispredictions by a factor of two.

Data Locality. In the second block, Table 3.1 presents performance counters to analyze data locality of the SPEs. Stream processing workloads usually access each input record only once, which causes a relatively high number of data-related cache misses. As shown in Table 3.1, Streambox and Flink induce the highest number of data cache misses across all cache levels. Additionally, Streambox causes 29x more TLB-D misses than any other SPE. These results indicate that Streambox and Flink cause more memory accesses for the same input data and that the utilized data layout and access patterns are sub-optimal. In contrast, Saber directly processes raw data, which causes fewer cache misses across all cache levels. However, Saber still causes at least 3.4x more L1 cache misses, 2x more L2 misses, and 6x more LLC cache misses compared to Grizzly. Both versions of Grizzly cause significantly fewer cache misses compared to all other SPEs and achieve higher data locality. As a result, the access latencies for records decrease, which leads to a significant speedup. Furthermore, Grizzly causes at least 10x fewer TLB-D misses compared to Saber. The high data locality of Grizzly highlights the benefit of direct data accesses on raw data without serialization, data copying, or object allocation overhead. The most efficient data locality is achieved by Grizzly^{++} that stores window state in a dense array, which results in 2.5x fewer L1 and 3.1x fewer L2 cache misses. These results are in line with the findings by Zeuch et al. [287].

Code Locality. In the last block, Table 3.1 shows performance counters related to code efficiency and locality of the SPEs. Overall, Streambox, Saber, and Flink execute at least 8x (Grizzly) and 27x (Grizzly^{++}) more instructions per input record. This highlights that Grizzly's code generation results in a very compact and CPU-friendly code. Furthermore, adaptive optimizations of Grizzly^{++} reduce the number of executed instructions by up to 3x. The results for instruction cache misses reveal, that Grizzly overall archives a much higher code locality. Flink and Streambox cause the most instruction cache misses, and many TLB-I misses.

In contrast, Saber causes 10x fewer instruction cache misses as a result of its micro-batch processing model. However, both versions of Grizzly cause basically no instruction cache misses and TLB-I misses per record. This indicates that the generated code fits entirely into the L1 instruction cache, and the generated instruction sequence is CPU-friendly.

Discussion Our analysis of resource utilization reveals that both Grizzly versions result in better control flow as well as higher data and instruction locality. Furthermore, exploiting data characteristics in Grizzly^{++} is vital to achieve peak performance. In contrast, for Flink, Streambox, and Saber, we observe inefficient memory utilization, which is caused by data serialization, object allocation, and the execution of inefficient and complex code. In sum, Grizzly's code generation for stream processing is essential to utilize resources of modern CPUs efficiently.

3.8 Related Work

We structure the related work into three areas: Work on stream processing engines, query compilation, and adaptive optimizations.

Stream Processing Engines. The first generation of SPEs laid the foundation to handle continuous queries over unbounded data streams [1, 2, 58, 61]. Due to growing data sizes and higher velocities, the second-generation of SPSs follow scale-out architectures while focusing on higher throughput, lower latency, and fault tolerance with exactly-once semantics [12, 32, 50, 54, 200, 266, 282, 286]. System S introduced optimizations for stream processing [109, 138]. In contrast to System S, Grizzly is a scale-up SPE that efficiently utilizes modern hardware. To this end, it fuses operators deeply together and eliminates any function calls between them, which still remain in System S.

Further examples for scale-up SPSs are SABER [164], Streambox [196], BriskStream [295], and Trill [56]. SABER focuses on hybrid stream processing on CPUs and GPUs. Streambox groups records in epochs and processes them for each operator in parallel. In contrast, Grizzly compiles queries into efficient code, which is executed using a task-based approach on a shared global state. Trill applies code generation techniques to rewrite user-defined functions to a block-oriented processing model over a columnar data layout. In contrast to Trill, Grizzly focuses on the fusion of multiple operators into one code block. BriskStream optimizes execution for NUMA hardware by distributing operations across NUMA-regions. In contrast, Grizzly follows a data-centric approach and executes operators on the NUMA nodes where the data is located. Furthermore, Grizzly fuses all operators into code without introducing unnecessary boundaries. Previous work showed that current SPEs, do not fully utilize the resources of modern hardware [287, 294]. Our work recognizes these limitations and proposes Grizzly, which generates highly efficient code. As a result, Grizzly outperforms state-of-the-art SPEs by at least an order of magnitude and reaches the performance of hand-optimized code.

Query Compilation. Query compilation for batch processing was extensively studied by Rao et al. [234], Krikellas et al. [166], and Neumann [202]. It was applied in many data processing systems [79, 102, 162, 166, 202, 203, 216, 256, 278]. Further work studied the support of user-defined functions [71, 253], query compilation for heterogeneous hardware [46, 225], efficient incremental view maintenance [10], the architecture of query compilers [6, 90, 161], and

the combination of compilation and vectorization [71, 194]. In this work, we complement the state-of-the-art by introducing query compilation for stream processing. Our technique enables the fusion of queries involving complex operations such as the window assignment, triggering, and aggregation. Recently, our query compilation approach was adopted by LightSaber [265], Darwin [31], and TilT [147]. LightSaber [265] proposes an novel approach to optimize concurrent sliding windows and TilT [147] introduces an high-level IR that models temporal operations. Both approaches could be integrated in Grizzly. Furthermore, Kroll et al. [167] proposed ARC to unify batch and stream queries, which could act as an input for Grizzly.

Adaptive Optimizations. In the data management community, adaptive optimizations have been extensively studied [21, 22, 256]. Răducanu et al. [232] proposed micro adaptivity by comparing the run-time of different operator implementations. Zeuch et al. [292], extended this approach by exploiting hardware counters to detect data properties like sortedness or operator order. In contrast, Dutt et al. [86] introduce explicit counters between operators to gather workload properties. In addition, recent work proposed adaptive optimizations for compilation-based engines for data-at-rest [130, 195, 245]. Furthermore, they also introduce adaptive compilation techniques to reduce compilation time [91, 162]. This line of work is orthogonal to our work, as we apply adaptive code optimizations to react to changing data characteristics in stream processing queries. In Grizzly, we combine these approaches to enable adaptive optimizations for stream processing. To this end, Grizzly monitors performance counters to detect changing data characteristics and generates instrumented code to collect detailed data statistics for optimization by using JIT-compilation. Additional work studied adaptive optimizations for stream processing [61, 109, 299]. These works mainly focused on the migration between query plans in a distributed setting. In contrast, Grizzly focuses on adaptive optimizations based on modern profiling techniques and query compilation to fully exploit modern hardware.

3.9 Conclusion

In this chapter, we transferred the concept of query compilation for data-at-rest queries to the operators and semantics of stream processing. We presented Grizzly, the first adaptive, compilation-based SPE that is able to generate highly efficient code for streaming queries. Grizzly supports streaming queries with different window types, window measures, and window functions. It utilized adaptive optimizations to react to changing data characteristics at runtime. To this end, we combine profiling techniques and apply task-based parallelization to fully utilize modern multi-core CPUs while fulfilling the ordering requirements of stream processing. Our experiments demonstrate that Grizzly outperforms the state-of-the-art SPEs by up to an order of magnitude due to better utilization of modern hardware.

With Grizzly, we laid the foundation for the efficient use of modern hardware in stream processing systems. Our contributions can either be integrated into current stream processing systems to improve their efficiency or serve as a blueprint for the development of the next generation of SPEs. Overall, Grizzly's high efficiency reduces the cost of streaming queries and makes stream processing more accessible to a wider audience of developers. To demonstrate the practicality of Grizzly, we leverage it as a foundation for NebulaStream.

3.9.1 Integration in NebulaStream

Based on Grizzly, we developed the compilation-based execution engine of NebulaStream. NebulaStream targets a wide set of data-processing workloads and hardware environments. To this end, we extended our Grizzly-based engine in the following dimensions:

Stream Slicing: We integrate general stream slicing [267, 268, 269] to improve the efficiency of overlapping sliding windows for aggregations and joins. Stream slicing assigns records to a single non-overlapping slice. This represents a specific portion of the stream and decouples the complexity of aggregations and joins from the actual window type. As a result, Grizzly can also reach a high throughput, even for overlapping sliding windows.

Event-Time Support: We extended Grizzly with support for event-time semantics. In this case, records are assigned to windows according to an $event_{ts}$ that is part of the record itself. As a result, records can be processed out-of-order, and the system requires a mechanism to detect if a window can be triggered. To this end, we adopted a punctuation-based approach and extended our lock-free window triggering strategy from Section 3.5.1 for watermarks.

Change detection for adaptive optimizations: We extended Grizzlies adaptive optimization strategy with a concept drift detector [124] to observe changes stream data characteristics. This enables NebulaStream to detect different types of changes across various data statistics, e.g., selectivity or runtime, without relying on hand-coded heuristics.

Additional Operators: We implemented for NebulaStream a wider set of data processing operators, including common arithmetic expressions, additional data types, and batch operators for aggregations and joins. To this end, we adapted known operators from literature to NebulaStream's architecture, e.g., Leis's parallelization-aware operator implementations [176].

Overall, Grizzly enabled NebulaStream to provide highly efficient data processing for various analytical workloads. The support of batch and stream processing in the same engine simplifies development and improves maintainability. Based on this, we will discuss the support UDF based-operators in the next chapter.

4

Query Compilation for Polyglot Queries

Over the last decades, the complexity of data processing workflows drastically increased. Today, interdisciplinary teams of data scientists, web developers, and application developers build complex data processing pipelines that combine different programming languages [9]. As a result, modern data management systems have to provide efficient support for complex data processing pipelines involving *polyglot queries*.

4.1 Introduction

Polyglot queries extend the relational algebra and combine relational operators with user-defined functions (UDFs). UDFs enable users to express arbitrary business logic in their preferred programming language [208], to leverage 3rd-party libraries [258], and to increase modularity and testability [25]. Today, many data processing engines support such polyglot queries [50, 238, 285], e.g., in the form of Java, Python, or JavaScript UDFs.

Although polyglot queries provide a large degree of freedom, their advantages come with a high performance penalty compared to traditional relational queries. This overhead is present across traditional database systems [62, 135, 165, 220] and big data processing frameworks [174, 228, 255]. The performance penalty originates from the underlying *impedance mismatch* between the declarative paradigm of SQL and the imperative paradigm of UDFs [233]. As a result, processing logic is scattered between the native execution engine and the language runtime, adding a level of indirection and preventing query optimization. Consequently, database experts recommend avoiding the use of polyglot UDFs whenever possible [94, 141, 174].

To cope with the inefficiency of polyglot queries, three different approaches have been proposed. First, multiple approaches study the translation of UDFs to semantically equivalent SQL statements [65, 84, 85, 89, 132, 134, 149, 233, 252]. However, these transformations are not always possible or lead to deep and complex operator trees, which are hard to execute efficiently [233]. Second, domain-specific languages for UDF-based data processing have been proposed [3, 14, 34, 111, 136, 170]. These languages enable advanced optimizations to improve the performance of UDF-based queries but lack the generality of languages like Python. Third, the direct embedding of UDFs in native query execution engines was extensively

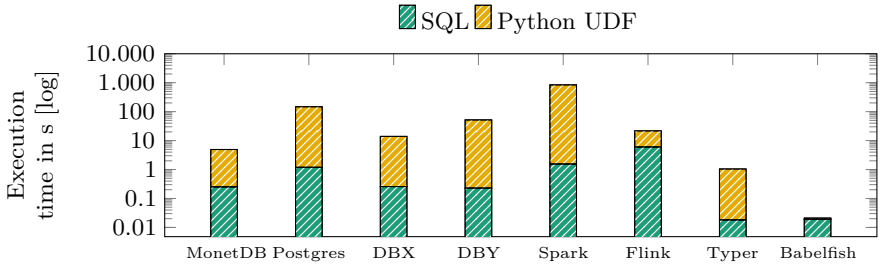


Figure 4.1: Overhead of TPC-H Query 6 with Python UDF.

studied [56, 76, 80, 145, 230, 238]. These approaches are applied in many systems but have two critical limitations. First, they only target the embedding of specific language runtimes. Thus, they require additional work to support multiple UDF languages. Second, they mainly focus on reducing the invocation overhead of the language runtime. Thus, they still suffer from data conversion or UDF execution overhead.

In Figure 4.1, we compare the overhead of polyglot queries across two open-source (MonetDB, Postgres) and two commercial (DBX, DBY) relational database systems, two big data engines (Spark, Flink), and a hand-written implementation based on Typer [156]. As a workload, we extend TPC-H Query 6 by a Python UDF similar to Ramachandra et al. [233]. For each system, we present the execution time with and without the Python UDF and illustrate the resulting performance overhead. All systems under test embed polyglot execution runtimes in their native execution engine, i.e., utilizing the third approach described above. Thus, they have to copy data between the native and the polyglot execution engine. Overall, we notice three aspects. First, we observe a significant overhead of the UDF-based query across all systems in contrast to the SQL implementation. Second, this overhead depends on the concrete execution engine and varies between 3.6x and 124x. Third, especially on systems with efficient SQL implementations, e.g., MonetDB and Typer, the Python UDF eliminates their performance advantage.

In this chapter, we propose Babelfish[1], a novel polyglot data processing engine. Babelfish unifies the execution of relational operators and UDFs in a single engine to overcome the performance limitations of current systems. To this end, Babelfish decouples the logical and physical representation of data and processing. This enables a holistic representation of polyglot queries and individual operators, independent of implementation languages. In particular, Babelfish addresses the impedance mismatch in polyglot queries in three steps. *1)* Babelfish combining relational operators and UDFs from different programming languages in one unified intermediate representation, the Babelfish-IR. *2)* Babelfish leverages this IR to apply traditional and new query optimizations to polyglot queries in a holistic and operator-agnostic fashion. *3)* Babelfish utilizes query compilation to translate polyglot queries into highly efficient code fragments. As a result, Babelfish significantly reduces the overhead of polyglot queries and outperforms all systems under test in Figure 3.1 by at least one order of magnitude. In summary, our contributions are as follows:

1. We define and formalize the foundational *impedance mismatch* between operators of polyglot queries.

[1]Babelfish: the oddest thing in the Universe. Effectively removes all barriers to communications between different cultures and races [4].

2. We introduce Babelfish, a novel query execution engine to improve the efficiency of polyglot queries.
3. We propose a unified intermediate representation for operators that is independent of their implementation language.
4. We introduce holistic optimizations for our query representation to eliminate the overhead of polyglot operators.
5. We evaluate Babelfish across different workloads and reach the performance of hand-written implementations.

The remainder of this chapter is structured as follows. First, we introduce a formal representation of polyglot queries in Section 4.2. Then, in Sections 4.3, we introduce a concept for the efficient execution of polyglot queries. Based on this concept, we describe Babelfish in detail in Section 4.4. In Section 4.5, we present optimizations to execute polyglot queries efficiently. Finally, we present our experiments in Section 4.6, related work in Section 4.7, and conclude in Section 4.8.

4.2 Polyglot Queries

Polyglot queries extend the relational algebra with UDFs and allow users to express processing logic in their preferred programming language.

Listing 4 shows an exemplary polyglot query that calculates the profit per user of a car-sharing business. The query combines three relational operators with two UDFs in Python and JavaScript. The `distance()` UDF embeds a 3rd-party library to calculate the distance between a start and end location. The `tripProfit()` defines the central business logic to compute the profit for a particular trip. Both UDFs consume input records as native data types of their programming languages, perform computations, and produce results.

In the remainder of this section, we use the previous example to define polyglot queries formally. This enables us to identify and address the structural limitations of such queries in current systems. First, Section 4.2.1 introduces a formal description of the structure of polyglot queries and individual operators. Based on this, Section 4.2.2 studies the data exchange between polyglot operators.

4.2.1 Modeling Polyglot Queries

Polyglot queries combine relational operators and UDFs in different programming languages. To derive a formal definition, we extend the tree-structured query representation of traditional data processing systems. Following Neumann et al. [204], we represent polyglot queries as trees of operators. A *query tree* is a directed, acyclic graphs $G = (V, E), |E| = |V| - 1$ with one root node $v_0 \in V$, such that all $v \in V \backslash \{v_0\}$ are reachable from v_0. Each vertex of the tree represents a polyglot operator PO_i that exchanges data via an edge with operator PO_j. Figure 4.2 illustrates the query tree of Listing 4 that exchanges data across five polyglot operators, from the initial `scan` to the `sink`.

In general, *polyglot operators* represent an arbitrary computation step in a query, i.e., in the form of a relational operation, a UDF, or a combination of both. For instance, the projection in Figure 4.2, embeds the `tripProfit()` UDF. During query processing, polyglot operators

```
-- SQL Query --
SELECT sum(tripProfit(t))
FROM trips t
WHERE distance(t.start, t.end) > 0.5
GROUP BY t.user_id

# Python distance function
from haversine import haversine, Unit
def distance(start, end):
   return haversine(start, end, Unit.KILOMETERS))

// JavaScript profit function
function tripProfit(t){
    let price;
    if(t.date.before("2020-01-01")){
      price = t.duration * 0.5;
    } else {
      price = distance(t.start, t.end) * 0.3;
    }
    return t.hasVoucher ? price * -1: price;
}
```

Listing 4: Polyglot Example Query.

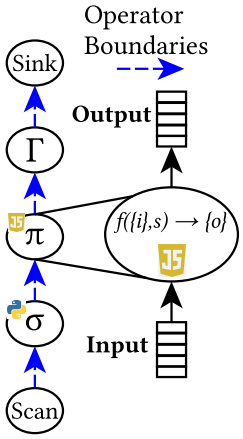


Figure 4.2: Query Tree.

consume input data, execute arbitrary computations, may manipulate intermediate state, and produce output data. Based on this, we define polyglot operators formally as the following tuple: ($InputType\ \tau_i$, $OutputType\ \tau_o$, $StateType\ \tau_s$, $f(\{i\}, s) \rightarrow \{o\}$). Thus, an operator defines a function f that consumes input i of type $InputType\ \tau_i$, accesses a state s of type $StateType\ \tau_s$, and produces an output o of type $OutputType\ \tau_o$. Each type corresponds to a specific data type τ that is defined for the execution environment Γ of a particular operator. In other words, Γ represents the data types that an operator potentially can process. For relational operators, Γ corresponds to the types of the SQL standard [87] and for UDFs Γ is defined by the UDF language, e.g., the JavaScript type system.

Besides the description of the data types of i, s, and o we define their physical representation. For instance, a `Numeric` data type in Python has a different physical representation than a `Numeric` in JavaScript. To this end, we introduce the *native data representation (NDR)* of an operator. The NDR_{PO_i} captures the physical representation in which the operator PO_i receives and produces data. The presented model enables us to define arbitrary polyglot queries consisting of polyglot operators. In the next section, we extend this model with a representation of data exchange between operators.

4.2.2 Modeling Data Exchange

Data exchange among polyglot operators defined in different programming languages requires data conversion. In the example query from Listing 4, the JavaScript-based `projection` receives data from a Python-based `selection` operator. Thus, a system has to convert all input records to JavaScript objects before executing the `selection` operator. The necessity of this data conversion is a direct consequence of the *impedance mismatch* of polyglot queries and a source of inefficiency in current polyglot systems.

To formalize the data exchange between two operators, we extend our query model and introduce the *common data representation (CDR)*. In contrast to the NDR of an individual operator, the CDR describes the form in which data is represented between two connected

operators PO_i and PO_{i+1}. To exchange data, an operator has to convert data from its internal NDR to the CDR. Thus, data exchange between two operators OP_i and OP_{i+1} is a transformation from the NDR of OP_i to the NDR of OP_{i+1} via a CDR. In particular, this transformation maps each field in the NDR to a corresponding field in the CDR ($NDR_{OP_i} \rightarrow CDR \rightarrow NDR_{OP_{i+1}}$).

In general, we can identify three data exchange types. In a *two-sided native* data exchange the NDR of both operators is equal to the CDR ($NDR_{PO_i} = CDR = NDR_{PO_i+1}$). Thus, both operators exchange data without any transformation. This corresponds to data exchanges between two built-in operators. In a *one-sided native* data exchange, the NDR of only one operator corresponds to the CDR ($NDR_{PO_i} = CDR \neq NDR_{PO_i+1}$). Thus, a transformation between its CDR and the NDR is required. One example is the embedding of Java UDFs, which requires the deserialization of Java Objects from raw data [108, 238]. In a *foreign* data exchange, both operators transform their NDR to the CDR by serializing and deserializing intermediate data ($NDR_{PO_i} \neq CDR \neq NDR_{PO_i+1}$). For example, one operator writes its output data to an intermediate format, e.g., CSV or Arrow, which another operator consumes. In the following, we utilize this model to describe the data exchange among polyglot operators.

4.3 The Case for Polyglot Queries

Many data processing systems support polyglot queries, e.g., MonetDB [230], Postgres [226], Exasol [189], Impala [278], or Flink [50]. These systems embed third-party language runtimes into their execution engines. Thus, they convert data from their internal data format to the NDR of the language runtime. Consequently, the data exchange between the execution engine and the language runtime, results in substantial performance overhead, as we saw in Figure 3.1. In the following, we first identify and analyze the main bottlenecks of polyglot query execution in current systems, see Section 4.3.1. Based on our analysis, we formulate design principles for a polyglot execution engine that addresses these bottlenecks, see Section 4.3.2.

4.3.1 Bottleneck Analysis for Polyglot Queries

In contrast to relational queries, polyglot queries introduce in many systems a boundary between the execution engine and the polyglot runtime. This boundary requires the following three additional processing steps for executing a polyglot query. *1)* The system has to transform each data record to the NDR of the polyglot operator, in order to make it accessible by the UDF. *2)* The system has to hand over the execution to the language runtime, whereby data crosses the operator boundary. *3)* The system receives processing results from the language runtime and has to translate them back into the system's NDR. These additional processing steps introduce three bottlenecks that limit the efficiency of polyglot queries.

Bottleneck 1: Runtime Invocation. The execution of polyglot operators requires the invocation of a language runtime, e.g., the JVM over JNI [183]. Each invocation causes a substantial performance overhead due to virtual function calls that decrease code locality [238]. Research proposed to reduce the number of runtime invocations by batching the invocation of polyglot operators [230, 238]. However, calling the runtime is still required and decreases code efficiency. Furthermore, it introduces overhead for data exchange and data conversion.

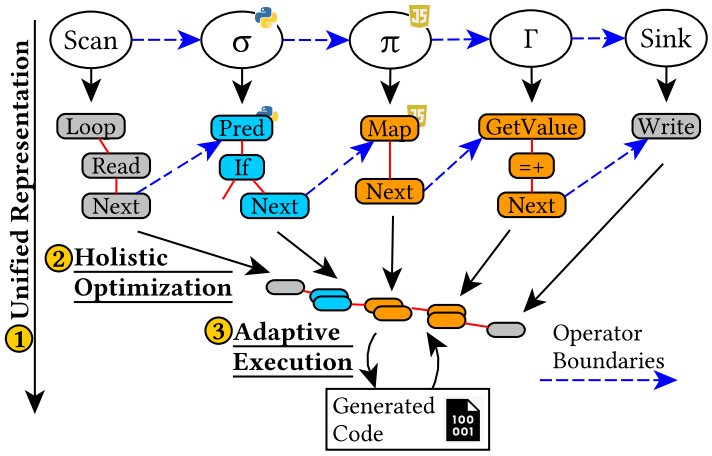


Figure 4.3: Polyglot query representation of Listing 4.

Bottleneck 2: Data Exchange. The boundary between the execution engine and the language runtime introduces data exchange. Before invoking a polyglot operator, the execution engine has to allocate memory, serialize intermediate results, and deserialize them in the language runtime. As a result, the engine introduces data copies that decrease data locality and memory efficiency.

Bottleneck 3: Data Conversion. Polyglot operators define custom data types, see Section 4.2. This requires the execution engine to convert intermediate records from the CDR to the NDR of a particular operator before invoking it. Thus, the execution engine has to access each field and convert it to the target data type, which introduces additional computations and data copies. Especially, the translation of complex data types, e.g., Point, Date, or Text, introduces a significant translation overhead. As a result, code efficiency decreases.

In general, operator boundaries in polyglot queries introduce function calls, data exchange, and data conversion that decrease code and memory efficiency. Consequently, the design of a new polyglot query execution engine should address these bottlenecks.

4.3.2 Efficient Polyglot Query Execution

Section 4.3.1 has shown that operator boundaries introduce several bottlenecks that reduce the efficiency of polyglot queries in current systems. To address these bottlenecks, we define three design goals, which mitigate the overhead of operator boundaries. As a running example, we visualize these design goals in Figure 4.3 using the query from Listing 4. In general, we aim for a holistic *representation* ①, *optimization* ②, and *execution* ③ of polyglot queries.

Design Goal ①: Unified Representation. Polyglot queries combine diverse operators that follow unique semantics for the representation of data and computations. For instance, the query depicted in Figure 4.3 combines relational operations with Python and JavaScript UDFs. To analyze and optimize polyglot queries across operator boundaries, it is essential to represent operators in a unified intermediate representation (IR). Such IR has to model the unique properties of diverse polyglot operators and the data exchange among them in a common representation. Furthermore, the IR should represent queries across different levels of abstraction to support optimizations, i.e., from the initial logical query plan down to the final machine code.

Design Goal ②: Holistic Optimization. The unified representation of polyglot queries (**DG1**) enables the holistic optimization of queries and operators independent of their definition language. Based on that, optimizations can address the query structure (e.g., reordering or operator fusion) as well as the implementation of individual operators (e.g., vectorization or predication). In particular, for polyglot queries, the execution engine can analyze and optimize data exchange and conversion between individual operators to mitigate the bottlenecks discussed in Section 4.3.1. For example, the optimizer could eliminate all operator boundaries for the query in Figure 4.3.

Design Goal ③: Adaptive Execution. The holistic optimization of polyglot queries (**DG2**) enables the mitigation of the operator boundary overhead. However, polyglot queries often involve dynamic languages, e.g., Python and JavaScript, that are hard to analyze and optimize before execution [39]. Especially, the optimization of the data exchange between operators requires knowledge of the operator's input and output types, which is only available at runtime. Consequently, for the efficient execution of polyglot queries involving dynamic programming languages, it is required to support adaptive optimizations at runtime.

Overall, these design goals are independent a of concrete implementation and could be incorporated into any engine to improve the efficiency of polyglot queries. Thus, they represent the first step towards efficient polyglot query execution.

4.4 Babelfish

This section introduces Babelfish[2], our novel data processing engine for polyglot queries. Babelfish applies the design principles from Section 4.3 to mitigate the overhead of polyglot query execution.

In particular, Babelfish introduces the Babelfish-IR as a unified representation of polyglot queries (**DG1**), performs holistic optimizations across operators (**DG2**), and applies just-in-time query compilation (**DG3**) to generate efficient data-centric [202] machine code. This enables, Babelfish to support polyglot queries, which combine relational operators and stateful Java, Python, and JavaScript UDFs, seamless and efficient in a single engine. To achieve this, Babelfish applies traditional database optimizations to the problem of polyglot queries and specializes query processing to specific requirements of polyglot queries. Furthermore, Babelfish relies on Truffle [281] and Graal [82] as technological foundation for just-in-time compilation and provides extensions for the efficient support of polyglot queries.

In the remainder, we present two aspects of Babelfish in detail. In Section 4.4.1, we introduce the Babelfish-IR and discuss the transformation of polyglot queries from the logical query plan to the final machine code. In Section 4.4.2, we introduce `BFRecords` to unify the data exchange among operators.

4.4.1 Query Representation

In Section 4.3.2, we have stated the desideratum of a unified and language independent representation of operators (**DG1**). To achieve this design goal, Babelfish introduces the Babelfish-IR as a unified intermediate representation of built-in operators and UDFs. Figure 4.4

[2]The source code is available at https://github.com/TU-Berlin-DIMA/babelfish.

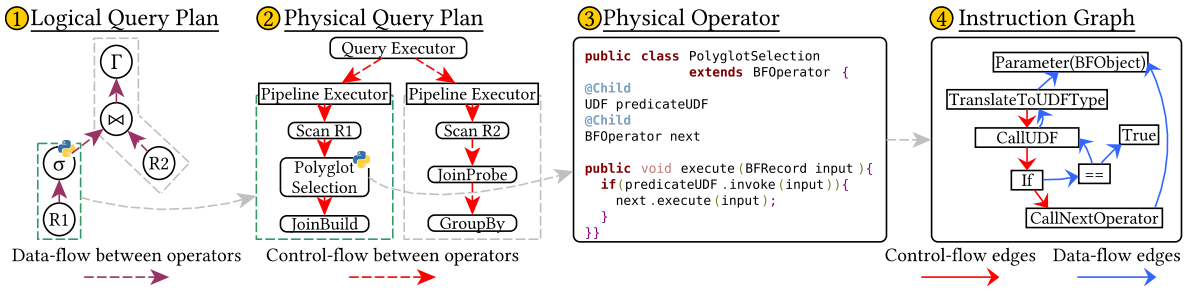


Figure 4.4: Multi-level Babelfish-IR for example query.

illustrates the Babelfish-IR for a polyglot query consisting of a `selection`, a `join`, and an `aggregation`. The Babelfish-IR represents queries in four levels of abstraction, the *Logical Query Plan* 1 (see Section 4.4.1.1), the *Physical Query Plan* 2 (see Section 4.4.1.2), a set of *Physical Operators* 3 (see Section 4.4.1.3), and the *Instruction Graph* 4 (see Section 4.4.1.4).

4.4.1.1 Logical Query Plan

The first abstraction level of the Babelfish-IR 1 represents queries as logical query plans (*LQPs*). LQPs combine relational operators and UDFs in a unified operator tree, whereby data flows from the leaves to the root (illustrated with dashed-purple arrows). For UDFs, Babelfish differentiates between *standalone UDFs* that implement complete operators, i.e., they receive and produce records, and *embedded UDFs* that extend relational operators, e.g., the selection in Figure 4.4 embeds a Python-based UDF as a predicate. From the LQP, Babelfish derives physical operators, segments the tree into pipelines (illustrated with dashed boxes), and creates the PQP.

4.4.1.2 Physical Query Plan

The second abstraction level 2 of the Babelfish-IR represents the query as a physical query plan (*PQP*). The PQP represents the control-flow (illustrated with dashed-red arrows) between individual physical operators of a query. The query executor at its root, manages the query execution, and invokes individual pipelines sequentially. Each pipeline starts with a scan over the data source, invokes a sequence of operators, and terminates with an operator that materializes results, i.e., a *pipeline breaker* like a join build or an aggregation. Thus, Babelfish follows a data-centric push-based execution model where data is pushed from the root of a pipeline, e.g., the scan, to the leaf operator, i.e., the pipeline breaker.

To implement the PQP, Babelfish leverages Truffle [281]. Truffle provides an implementation framework for programming languages, e.g., GraalJS [211] and Graal-Python [210]. Based on Truffle, Babelfish represents the PQP, provides implementations of relational operators, and handles the interaction between operators and UDFs.

4.4.1.3 Physical Operators

The third abstraction level 3 of the Babelfish-IR represents individual physical operators. Each operator corresponds to a Java implementation using the Truffle framework. In particular, Babelfish differentiates between one of three operator kinds, i.e, *built-in*, *UDF-based*, or *primitive* operators.

Built-in Operators: Built-in operators are provided by Babelfish and refer to the implementation of specific logical operators, e.g., `JoinBuild` for the build-side of a relational `join`. Similar to the Volcano Model [117], built-in operators define three functions, `open()`, `close()`, and `execute()`. `Open()` and `close()` initialize and finalize the operator state. In contrast, `execute()` defines the processing logic of the operator and receives input records.

UDF-based Operators: Babelfish represents standalone and embedded UDFs as individual operators in the PQP. In Figure 4.4, we sketch a selection operator that embeds a UDF as a predicate. This operator receives a record, evaluates the UDF, and executes the next operator if the predicate matches. For the execution of UDFs, Babelfish relies on Truffle-based JavaScript [211] and Python [210] implementations. Each language implementation defines special Truffle nodes to capture language semantics. To integrate the UDFs with Babelfish's type system and semantics, Babelfish wraps the invocation of UDFs and handles the data transfer between build-in operators and UDFs.

Primitives: The Babelfish-IR introduces primitives to generalize common data processing operations across different operator implementations. For instance, the physical `JoinBuild` and `GroupBy` operators use the same primitive to access a hash-map. This reduces complexity and improves the maintainability of operator implementations, as complex operators assemble multiple primitives.

In general, physical operators unify the implementation of operators independent of their kind. Based on this, Babelfish constructs the instruction graph to enable holistic optimizations.

4.4.1.4 Instruction Graph

The fourth abstraction level ④ of the Babelfish-IR represents the implementation of physical operators on the instruction graph. In contrast to the PQP, the instruction graph represents individual instructions or specific operations, e.g., memory accesses. As a foundation, Babelfish relies on the Graal IR [82], which is a graph-based IR for generic programs in static single assignment (SSA) form [73]. Thus, each node represents a specific operation, which may depend on input nodes and produces at most one value. Between individual operations, the graph captures the control-flow (red arrows downwards) and the data-flow dependencies (blue arrows upwards). Babelfish extends the Graal IR with custom nodes to represent primitive operations, e.g., the `BFLoad` node loads a record field and takes knowledge about the underlying memory layout into account, or the `StartTransaction` node protects the entry point of a critical section. In Figure 4.4, we show in ④ the instruction graph of the UDF-based selection operator. It consists of a conditional branch and two function calls, one to the UDF and one to the next operator.

The instruction graph enables Babelfish to analyze and optimize individual operator implementations holistically across built-in operators and UDFs. During the optimization phase (**DG2**), Babelfish modifies the instruction graph and performs several optimizations, e.g., operator fusion, loop unrolling, or elimination of intermediate values (see Section 4.5). During execution, Babelfish applies just-in-time compilation (**DG3**) and translates the instruction graph to executable machine code using Graal. In general, the instruction graph provides fine-grained control over the final machine code, which is crucial to exploit modern hardware efficiently.

4.4.2 Data Representation

In the previous section, we introduced the representation of polyglot queries on our Babelfish-IR. During execution, operators process data in their respective NDR. For instance, the Python-based selection in Figure 4.4 operates on Python objects. To exchange data between arbitrary operators, Babelfish introduces `BFRecords` as a general CDR. `BFRecords` serves three different goals. *1)* `BFRecords` unifies data exchange among operators independent of their definition language. *2)* `BFRecords` supports different data types to enables a wide range of workloads. *3)* `BFRecords` decouples logical and physical representation of data. In the remainder, we first introduce `BFRecords` in detail (see Section 4.4.2.1). Based on this, we present `PolyRecords` for the data exchange among built-in operators and UDFs (see Section 4.4.2.2). Finally, we describe the mapping of `BFRecords` to a physical memory layout (see Section 4.4.2.3).

4.4.2.1 Babelfish Records

Babelfish introduces `BFRecords` to represent data exchange between operators. `BFRecords` are record types [53] that define a collection of fields and each field consists of a name and a type In the instruction graph, Babelfish represents accesses of individual fields with `BFReadField` and `BFWriteField` nodes. These nodes capture field information, i.e., the field index and the field type. This enables Babelfish to analyze and optimize the data exchange between operators, e.g., identifying if PO_i stores a value that PO_j reads. For the data types of `BFRecords` fields, Babelfish differentiates between *primitive*, *collection*, and *composed* types.

Primitive Types. Babelfish supports common primitive data types, e.g., Boolean, Int, or Double. These types build the foundation for composed data types and are crucial for primitive expressions, e.g., `linenumber` > 2. Operations on these types usually directly correspond to hardware operations in the final machine code.

Collection Types. Babelfish supports multidimensional collections of values in the form of fixed-sized `arrays` and variable-length `lists`. To operate on these types, Babelfish introduces `LoadIndex`, `StoreIndex`, and `GetLength` nodes on the instruction graph. Furthermore, Babelfish represents text as a special collection type with efficient support for common text operations, e.g., `substring`.

Composed Types. Babelfish leverages compositions to define complex domain types, e.g., `date`, `numeric`, or `point`. Compositions consist of fields and allow the definition of type-specific operations. For instance, the `date` type is represented as and defines custom operations, e.g., `before(date)`, or `from(text)`.

In general, `BFRecords` enable Babelfish to represent different data types in a common data representation that is agnostic of the definition language of operators.

4.4.2.2 Polyglot Records

Before executing a UDF, Babelfish has to convert `BFRecords` to a data type defined by the particular UDF language, e.g., a Python object. To this end, Babelfish leverages Truffle's Foreign Message Interface [120] and wraps `BFRecords` in `PolyRecords`. `PolyRecords` integrate seamlessly with a particular UDF language and behave as native objects for a user. To reduce conversion overhead, Babelfish introduces two strategies. For primitive data

types, Babelfish substitutes accesses to `PolyRecord's` with corresponding instruction graph nodes, e.g., `BFReadField`. This enables the UDF to bypass data conversion and to operate on `BFRecords` directly. For complex types, e.g., collections or compositions, Babelfish applies duck-typing[3] and defines proxy data types. These proxies substitute the functionality of particular data types and redirect all operations directly to the underling `BFRecords` without any data conversion. For instance, Babelfish defines a proxy for Python `strings` that substitutes common operations, e.g., `substring`, or `split`.

As a result, `PolyRecords` enable users to rely on a familiar programming interface of native objects while it accesses intermediate data directly without any additional data conversion.

4.4.2.3 Physical Data Representation

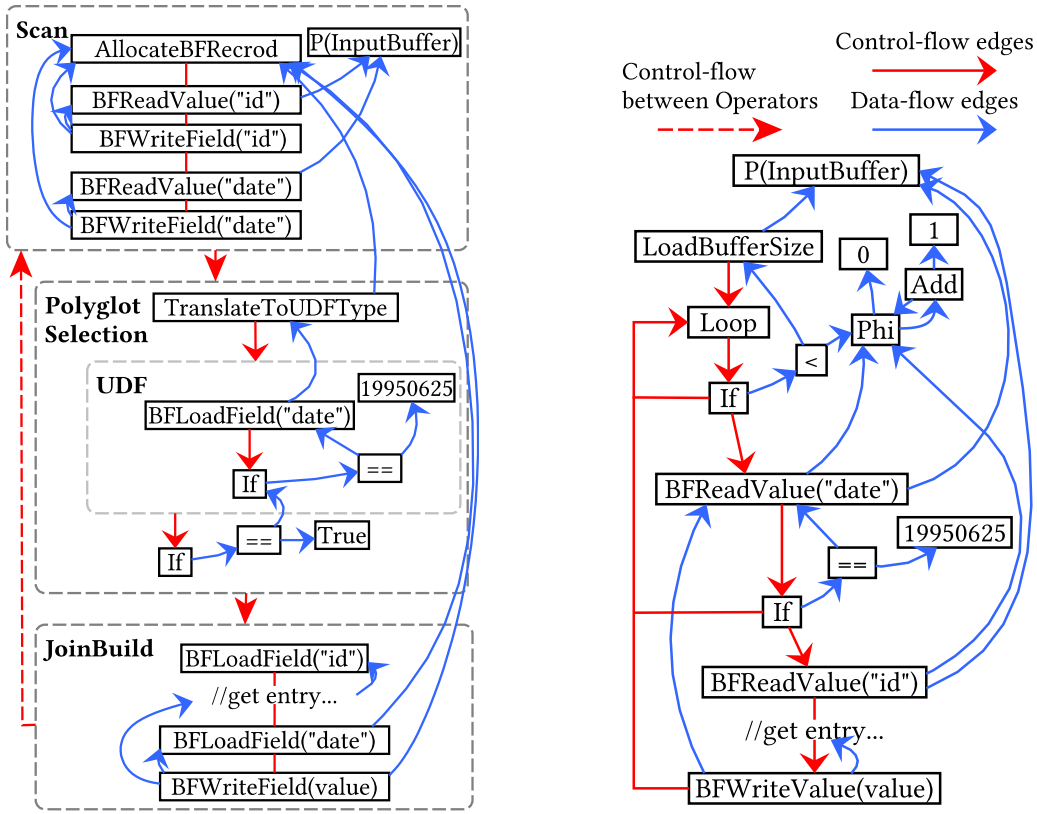
Operators in Babelfish exchange data in the form of `BFRecords`. Thus, they are independent of the physical data representation. This improves flexibility and maintainability as operators make no assumptions about the representation of data in memory. At pipeline boundaries, Babelfish (de)serializes `BFRecords` to and from memory according to a layout descriptor. This descriptor either corresponds to a third-party data format (e.g., CSV or Arrow) or is generated by Babelfish (following a DSM or NSM layout). For both cases, it defines the format in which the individual fields are stored in memory, how much space values occupy, and an access strategy. For generated layouts, Babelfish introduces special `ReadValue` and `WriteValue` nodes on the instruction graph. These nodes encapsulate particular field access information, e.g., for offset calculation, resulting in very efficient code for memory accesses.

In general, `BFRecords` decouple operators and UDFs from the physical data representation. During the optimization phase, Babelfish eliminates all `BFRecords` and access memory directly.

4.5 Optimizing Polyglot Queries

In the previous section, we have introduced our Babelfish-IR as a unified representation for polyglot queries. In this section, we leverage the Babelfish-IR to introduce holistic optimizations across built-in and UDF-based operators. In particular, we focus on eliminating the structural boundaries between operators in polyglot queries (**DG2**). To this end, we propose the following optimizations to mitigate specific limitations in common polyglot workloads. In Section 4.5.1, we present *operator fusion* to eliminate operator boundaries. Based on this, we propose three exemplary cross-operator optimizations that are possible with our Babelfish-IR and overcome bottlenecks of common polyglot queries. First, we present *polyglot predication* to optimize selective UDFs in polyglot queries in Section 4.5.2. Second, we introduce *latch-reduction* to improve the efficiency of stateful polyglot operators in concurrent environments in Section 4.5.3. Third, we present *workload specializations* to improve the efficiency of raw and textual data processing in Section 4.5.4.

[3]Uses duck-test to determine if a function can be called on an object. "If it looks like a duck and quacks like a duck, it must be a duck" [99]



(a) Initial IR **(b)** IR after operator fusion

Figure 4.5: Illustration of the elimination of operator boundaries based on the instruction graph for the build pipeline of Figure 4.4.

4.5.1 Eliminating Operator Boundaries

In Section 4.3.1, we have outlined that boundaries between operators are one of the major bottlenecks of polyglot queries. These boundaries introduce function calls, branches, and redundant data copies that lead to decreased instruction and data locality. To address these issues in relational data processing engines, Neumann [202] proposed a data-centric execution strategy. To this end, he leverages code generation and fuses multiple operators into efficient code blocks. As a result, data may reside longer in CPU registers, without any indirection. This technique has been utilized in multiple data processing systems and demonstrated high-performance benefits [46, 125, 203, 204, 217].

In contrast to traditional, code generation-based operator fusion, we leverage our Babelfish-IR and propose *IR-based operator fusion*. This has two main advantages. First, IR-based operator fusion is independent of the language used to define the operators. Thus, Babelfish can fuse pipelines of built-in operators and arbitrary UDFs. Second, IR-based operator fusion manipulates the Babelfish-IR directly and generates no intermediate source code or external IR. This improves maintainability and increases the impact of further optimizations.

In general, Babelfish's IR-based operator fusion aims for two goals, the elimination of function calls between operators and the elimination of intermediate allocations. In Figure 4.6, we illustrate operator fusion for the build pipeline from Figure 4.4. The pipeline consists of three physical operators, a `Scan`, a `PolyglotSelection`, and a `JoinBuild`. In Figure 4.5a, we visualize the initial instruction graph and the boundaries between operators (boxes). To

eliminate these boundaries, Babelfish performs IR-based operator fusion within three steps: *specialization*, *inlining*, and *scalar replacement*.

1) Specialization. The initial instruction graph of an operator corresponds directly to its implementation. Thus, it may capture execution paths that are never taken. For example, a division requires different implementations for each input data type. This increases instruction graph complexity and hinders optimizations.

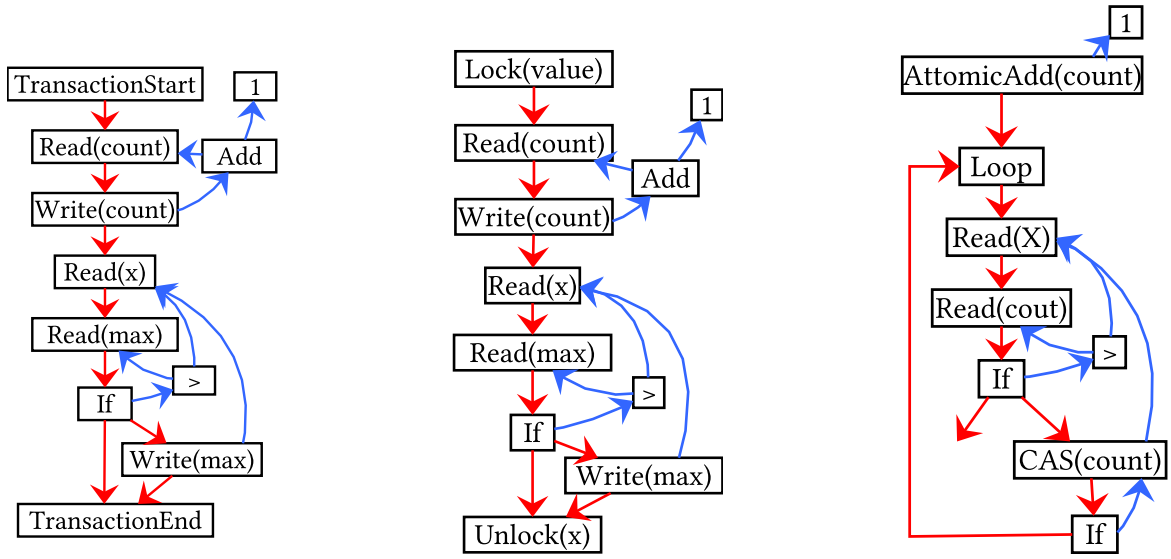
To reduce this complexity, Babelfish leverages Truffle's partial evaluation [283]. Partial evaluation combines the code of a generic program with runtime constant input data to create a specialized program [105, 150]. This specialized program produces the same output but eliminates all computations that depend on the constant input. For example, partial evaluation specializes divisions to `Shift` instructions if the divisor is a constant power of 2 [279].

Babelfish applies partial evaluation to specialize operator implementations according to runtime parameters, e.g., the PQP structure, operator properties, or the physical data layout. In particular, specialization enables Babelfish *1)* to analyze and de-virtualize function calls between operators, *2)* to bind data types to variables according to the data schema, and *3)* to optimize the data accesses for a specific physical data layout. As a result, specialization reduces the complexity of Babelfish's instruction graph.

2) Inlining. In this step, Babelfish utilizes the specialized instruction graph and inlines all function calls between operators within a pipeline. For the pipeline in Figure 4.5b, Babelfish creates a compact instruction graph that contains all connected operators. Thus, the pipeline follows a data-centric execution model and contains no function calls. Within individual operators, Babelfish relies on the inlining heuristics of Graal [180], e.g., to inline calls to 3rd-party libraries within a UDF. As a result of this step, Babelfish fuses the individual operators and derives a unified instruction graph without boundaries between operators.

3) Scalar Replacement. In the previews steps, Babelfish has reduced the complexity of the instruction graph by applying specialization and inlining. However, the instruction graph still contains intermediate objects, e.g., `BFRecords`. This causes unnecessary memory allocations and data copies. To avoid those, Babelfish applies *Scalar Replacement* [213, 283]. Scalar replacement eliminates intermediate objects by rewriting accesses to objects with local variables. This removes allocations and intermediate objects. In contrast to a general-purpose compiler, Babelfish can guarantee that allocations within a pipeline never escape the scope, i.e., they can not be stored in external states except the operator state variables. Consequently, scalar replacement eliminates all intermediate `BFRecords` during compilation. Furthermore, Babelfish rewrites all field accesses with pointers to the actual memory location. As shown in Figure 4.5b, scalar replacement eliminates the intermediate `BFRecords` and replaces all `BFReadField` nodes with `BFReadValue` nodes. As a result, operators directly access raw memory, data reside longer in CPU registers, and the code follows a data-centric structure [202].

Overall, IR-based operator fusion eliminates the boundary between built-in operators and UDFs, independent of their definition language. To this end, Babelfish specializes operators, inlines function calls, and eliminates intermediate objects. As a result, processing pipelines perform no function calls, allocate no intermediate objects, and access memory directly.



(a) Initial IR **(b)** Locking **(c)** Atomics

Figure 4.6: Illustration of latch-reduction for a stateful UDF.

Additionally, operator fusion in Babelfish enables further cross-operator optimizations to improve the efficiency of polyglot queries, e.g., *predication* and *loop unrolling*.

4.5.2 Optimizing Polyglot Predicates

Predicates are an important primitive in relational operators and UDFs to filter or manipulate data depending on conditions. In UDFs, filter conditions are usually implemented using a conditional branch instruction. Depending on the branch selectivity, miss-predictions may occur. This can induce a high runtime overhead [291]. To mitigate this overhead in relational data processing engines, research proposed branch-free operator implementations, using *predication* [47, 239, 297]. However, this technique requires the modification of all operator implementations in a pipeline. Consequently, it is challenging to apply predication on polyglot queries with UDFs. To overcome this challenge, we introduce *IR-based predication*.

IR-based predication. Our approach identifies and eliminates predicates across operators directly on the instruction graph in two steps. First, Babelfish identifies all predicates and utilizes profiling information to derive the filter selectivity. Second, Babelfish eliminates the predicate and leverages predicated CPU instructions to rewrite data manipulations. Predicated instructions allow the CPU to perform operations depending on the outcome of a condition, e.g., the `CMOV` instruction only performs a `MOV` if a condition is valid.

Overall, IR-based predication demonstrates the benefits of Babelfish-IR for query optimization. It leverages profiling information to eliminate conditional branches, independent of the definition language of operators. Furthermore, this reduces the control-flow complexity, which is beneficial for further compiler optimizations (see the experiment in Section 4.6.3.4).

4.5.3 State Management

Babelfish supports stateful operators and UDFs to model complex business logic. These operators maintain and manipulate local state that lives beyond a single operator invocation.

Thus, the execution engine has to ensure the correct concurrent execution of such operators. To this end, many systems use external state-backends, e.g., RocksDB, to support stateful operators [16, 50]. The state-backend coordinates concurrent state accesses and decouples concurrency management from data processing. This improves maintainability, but often hinders optimizations. In contrast, Babelfish models state accesses directly on the instruction graph (see Figure 4.6a). `TransactionStart` and `TransactionEnd` nodes mark the critical section of a state manipulation. For all operations within this critical section, Babelfish uses latches to guarantee mutual exclusion. During execution, each thread acquires a latch to protect the state variable from concurrent manipulations, as shown in Figure 4.6b. With high contention, latches may cause a significant overhead [42]. To mitigate this, Babelfish introduces *latch-reduction* and replaces latches with atomic operations.

Latch-reduction. To eliminate latches, Babelfish manipulates the instruction graph in two steps. In the first step, Babelfish analyses the critical section and extracts all distinct state modifications. Distinct state modifications do not depend on a common input value and thus Babelfish can optimize them independently. For example, the instruction graph in Figure 4.6b performs two distinct modifications, i.e., the `ADD` of `count` and the conditional `Write` on `max`. In the second step, Babelfish translates each data manipulation into an atomic operation (see Figure 4.6c). For arithmetic operations, Babelfish creates individual nodes, e.g., `AtomicIncrement`. For complex control flow, Babelfish generates compare and swap loops. If latch reduction is not possible, Babelfish falls back to spin-locks.

Overall, *latch-reduction* improves the execution performance of stateful UDFs under contention (see the experiment in Section 4.6.3.3).

4.5.4 Workload Specialization

Modern data analytic workloads drastically differ from traditional relational queries. They often contain text-heavy computations [277] or directly process raw data [154]. To support these workloads in a polyglot execution engine efficiently, Babelfish relies on the strict separation of physical and logical data processing. This enables Babelfish to optimize workloads independent of operator definition languages. In the following, we present Babelfish's handling of textual data in Section 4.5.4.1 and the support of raw data in Section 4.5.4.2.

4.5.4.1 Text Processing

UDFs often analyze or manipulate text values, e.g., word-count, tokenization, or n-gram computation [170]. In languages like Java or Python, these operations cause a high overhead as they represent text as immutable `String` objects. For instance, `String` concatenation in a Java UDF performs three memory copies, i.e., *1)* the creation of a `String` object from the input, *2)* the concatenation with another `String`, and *3)* the materialization to the output. To mitigate this overhead, Babelfish introduces `PolyglotRopes` to enable efficient text manipulations.

Polyglot Ropes. A Ropes is a data-structure to represent text as a tree of text fragments and operations [36]. Text manipulations result in tree transformations, e.g., the concatenation of two text fragments results in a `concat` node with the fragments as children. Babelfish leverages this concept and defines `PolyglotRopes` to represent text across operators. `PolyglotRopes` differentiate between leaf ropes and operation ropes. Leaf ropes reference fragments of the

overall text at a particular location, i.e., a `PointerLeaf` references a memory region on the input data, or a `ConstantLeaf` references a constant text sequence. In contrast, operation ropes express text manipulations, e.g., the concatenation of two ropes results in a `ConcatRope`. This allows Babelfish to evaluate text operations lazily and materializes ropes only if required, e.g., if the text is written to the output. As a result, Babelfish performs the concatenation of two texts with only a single data copy, i.e., it reads both input texts only once and writes them directly to the output without any allocations.

Overall, `PolyglotRopes` reduce data copies by executing text operations lazily. Combined with `PolyRecords`, Babelfish can substitute text operations across arbitrary UDF languages with `PolyglotRopes`. As a result, this optimization improves the performance of text-heavy polyglot workloads significantly (see Section 4.6.3.5).

4.5.4.2 Raw Data Processing

Polyglot workloads often directly process raw files in third-party data formats, e.g., CSV or Arrow [101]. Before processing, most systems convert raw data to an internal data format. This requires data parsing and materialization, which induces a high overhead [142]. To mitigate this, Babelfish proposes *lazy parsing* and interleaves parsing with polyglot query processing.

Lazy Parsing. In Section 4.4.2, we have introduced `BFRecords` to separate the physical data representation from the operator implementations. `BFRecords` also enable Babelfish to process raw data transparently. In the `Scan` operator, Babelfish parses the raw data, initiates `BFRecords`, and passes them to the processing pipeline. Then operator fusion eliminates the intermediate `BFRecords` and interleaves parsing and query processing. On the resulting IR, Babelfish performs two additional optimizations. First, Babelfish eliminates the parsing for fields that are not accessed. This is beneficial for workloads that only access a subset of the raw data. Second, Babelfish delays the parsing of individual fields. This is beneficial as field accesses often depend on conditions. Furthermore, it reduces the distance between data parsing and access, which reduces register pressure.

Overall *lazy parsing* interleaves raw data parsing and query processing. This enables Babelfish to eliminate or delay the parsing of particular fields. As a result, Babelfish is able to significantly improve query processing performance over raw data (see Section 4.6.3.6).

4.6 Evaluation

In this section, we experimentally evaluate different aspects of Babelfish. We introduce our experimental setup in Section 4.6.1. After that, we conduct two sets of experiments. In Section 4.6.2, we compare the performance of Babelfish across multiple workloads to state-of-the-art data processing systems. In Section 4.6.3, we perform micro-experiments to study specific aspects of Babelfish.

4.6.1 Experimental Setup

Throughout our evaluation, we use the hardware and software configurations that are described in Section 4.6.1.1 and run the workloads and datasets that are described in Section 4.6.1.2.

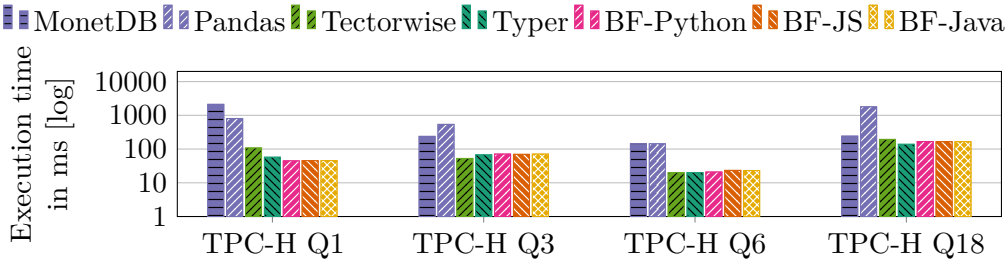


Figure 4.7: Comparison of Babelfish with hand optimized baselines on TPC-H queries.

4.6.1.1 Hardware and Software

We execute all experiments on an Intel Xenon Gold 6126 processor with 2.6 GHz and 12 physical cores. Each physical core has a dedicated 32 KB L1 cache for data and instructions. Additionally, each core has 1MB L2 cache and all cores share a 19.25 MB L3 cache. The test system consists of 755 GB of main memory and runs Ubuntu 20.04. The C++ implementations are compiled with GCC 9.2 and Babelfish's implementation runs on the community edition of GraalVM 20.3. If not stated otherwise, we execute all measurements using a single processing thread.

4.6.1.2 Workloads

Throughout this evaluation, we use the following datasets (stored in memory in a columnar format). To evaluate the OLAP performance, we use queries from the *TPC-H* with a scale factor of one, which results in ~1GB of data For data-science queries, we use the *Airline On-Time Performance Dataset* [207], which was used in multiple publications to assess the performance of big data systems on common data science workloads [170, 227, 273]. This dataset contains data about flights between 2018-2020 and additional information, e.g., departure time or origin/destination. After cleaning, it contains ~2GB of data.

4.6.2 System Comparisons

In this set of experiments, we study the performance of Babelfish and representative data processing systems on relational queries (see Section 4.6.2.1), data science workloads (see Section 4.6.2.2), and UDFs that embed 3rd-party libraries (see Section 4.6.2.3).

4.6.2.1 Relational Workloads

In this experiment, we investigate Babelfish's performance across four TPC-H queries, i.e., Q1, Q3, Q6, and Q18. These queries represent different workload characteristics, e.g., aggregations or joins, and have been used before to assess the efficiency of data processing engines [156]. Similar to Ramachandra et al. [233], we replace operators with either Python, JavaScript, or Java UDFs within Babelfish. As baselines, we evaluate all queries without UDFs on MonetDB [40], a general-purpose DBMS, Pandas [191], a common data processing framework, and Typer/Tectorwise [156] as hand-optimized C++ implementations of the above queries.

Results. Figure 4.7 shows execution times of MonetDB, Pandas, Typer, Tectorwise, and Babelfish for the selected queries. Across all queries, Babelfish outperforms MonetDB/Pandas

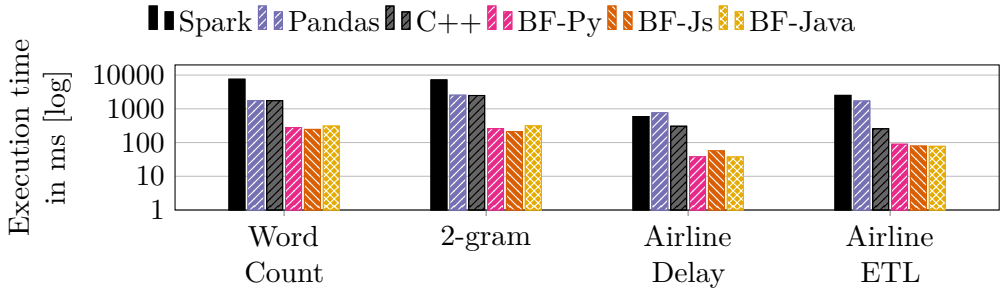


Figure 4.8: Comparison of Babelfish with hand optimized baselines across data science workloads.

by up to one order of magnitude and reaches similar performance to Typer and Tectorwise. In contrast to MonetDB and Pandas, Babelfish benefits from operator fusion and the elimination of intermediate results. In comparison to Typer and Tectorwise, our results are in line with the general observation of Kersten et al. [156]. For compute-heavy queries (e.g., TPC-H Q1) Typer and Babelfish benefit from the data-centric execution model that holds data within registers. As a result, they improve performance by up to 2.4x compared to Tectorwise. For join-heavy queries (e.g., TPC-H Q3) Tectorwise has a performance advantage of up to 1.3x compared to Typer, and Babelfish, as the vectorized execution model hides cache misses. Furthermore, our results show only a negotiable performance difference between individual UDF languages on Babelfish. This indicates an advantage of the Babelfish-IR as it allows optimizations across operator boundaries.

Summary. This experiment shows that Babelfish outperforms MonetDB and Pandas by up to an order of magnitude and reaches the performance of hand-written query implementations proposed by Kersten et al. [156]. Furthermore, we saw that the performance variations between different UDF implementation languages in Babelfish are negligible. As a result, Babelfish closes the gap between purpose programming languages and the performance of hand-written code.

4.6.2.2 Data Science Workloads

In this experiment, we examine the performance of Babelfish across two common data science building blocks i.e., word-count and 2-gram computation, as well as two real-world workloads, i.e., Airline Delay and Airline ETL. Following the implementation of Lara et al. [227], we implement each query as a sequence of Python, JavaScript, and Java UDFs. As a baseline, we evaluate the selected queries on Spark [285] and Pandas [191], as common data science frameworks, and a hand-written C++ implementation.

Results. In Figure 4.8 we observe that Spark induces the highest execution time. This is in line with previous observations that Spark utilizes the hardware resources of single-node setups poorly [90, 287]. In contrast, Pandas offloads computations to efficient C++ extensions and performs the word-count and 2-gram queries up to 4x faster than Spark. However, on the Airline queries, our hand-written C++ implementation outperforms Pandas by up to 6x as it reduces function calls and avoids intermediate materialization.

Babelfish applies operator fusion to reach the same code efficiency. As a result, Babelfish outperforms Spark and Pandas by up to one order of magnitude. In comparison to the hand-

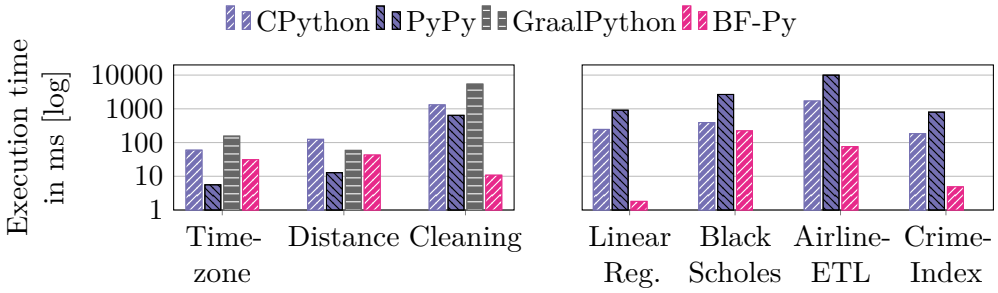


Figure 4.9: Comparison of Babelfish with hand optimized baselines across UDFs Embedding 3rd-Party Libaries.

written C++ baseline, Babelfish achieves a performance improvement between 3x to 11x due to workload specialization. Especially, the word count and the 2-gram computation cause many string allocations, i.e., each input record results in multiple output strings. Babelfish leverages `PolyglotRopes` to eliminate string intermediate materialization. Consequently, Babelfish even outperforms the hand-written C++ implementation by 10x on the word count and 2-gram query.

In general, we observe a slight performance variance (max. 20%) between Java, Python, and JavaScript. This is mainly caused by specific implementation artifacts of the particular language. For instance, for iterative computations, e.g., word count, the Truffle implementations of Python, Java, and JavaScript result in different code after loop unrolling was applied. We expect that such performance differences will vanish in the future if Truffle becomes more mature. Overall, Babelfish outperforms all baselines, regardless of the UDF language.

Summary. This experiment showed that Babelfish reaches high performance on data science workloads. Babelfish benefits from the optimization of polyglot queries, which fuse operators, eliminate allocations, and specialize operations within UDFs. As a result, Babelfish outperforms data science frameworks like Spark and Pandas.

4.6.2.3 Embedding 3rd-party libraries in UDFs

A common use case of UDFs is the embedding of 3rd-party libraries [258]. Therefore, we evaluate seven queries [214, 227] with Python UDFs that embed different libraries. The first three queries embed `Arrow`, `Haversine`, and `Re`, which are implemented purely in Python. In contrast, the remaining four queries leverage `NumPy`, and `Pandas`, which heavily use native Python extensions. To investigate Babelfish's impact, we use three baseline, i.e., CPython [98] (the standard Python runtime), PyPy [39, 262] (a high performance Python JIT-Compiler), and GraalPython (Babelfish's underlying Python runtime).

Results. Figure 4.9 shows that Babelfish outperforms CPython across all queries, whereby the speedups depends on the actual workload. Queries that embed pure Python libraries benefit from JIT compilation, i.e., PyPy outperforms the other Python runtimes by up to 5x. Due to the prototypical state of GraalPython, it can not to deliver a similar performance. In particular, calls into the standard library, e.g., for regular expressions or date calculation, cause a high-overhead in the current version. Babelfish mitigates this overhead with its efficient memory layout and optimizations for string processing.

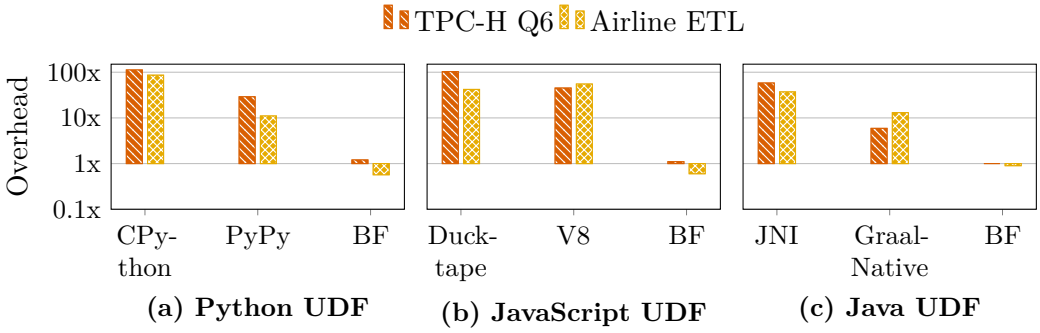


Figure 4.10: Overhead of language runtimes and Babelfish.

For queries that embed native Python extensions, JIT compilers have to (re)implement the C-API of CPython [100]. As a result, PyPy suffers from a high overhead, and GraalPython fails to execute the queries. This is a known problem of Python JIT compilers [257] and is a focus of the HPy project [263]. To mitigate this problem, Babelfish leverages the `BFRecords` to substitute calls into these libraries with build-in Babelfish functions. This approach is similar to Weld [214] and enables Babelfish to fuse operators, optimize data accesses, and eliminate intermediate materializations. As a result, Babelfish outperforms CPython by up to two orders of magnitude.

Summary. This experiment showed that Babelfish efficiently executes queries that embed 3rd-party libraries. Babelfish benefits from its efficient memory layout and can apply optimizations across the library code. Furthermore, Babelfish substitutes library calls to mitigate the limitations of GraalPython.

4.6.3 Micro Experiments

In the previous experiment section, we have demonstrated that Babelfish achieves high performance for end-to-end workloads. In this section, we focus on particular workload details that impact the performance of Babelfish. First, we evaluate the embedding of different language runtimes in a data execution engines, to validate Babelfish's core design principle in Section 4.6.3.1. Based on this, we study the effect of Babelfish's JIT compilation on execution performance and warm-up time in Section 4.6.3.2. Then, we evaluate the handling of stateful UDFs in multi-core environments in Section 4.6.3.3. Finally, we analyze Babelfish's performance for specific workloads, i.e., string operations in Section 4.6.3.5 and raw data processing in Section 4.6.3.6.

4.6.3.1 Language Runtimes

In our initial experiment in Figure 3.1, we revealed a high overhead of polyglot queries on modern data processing systems. Based on this observation, we derived that a unified representation of polyglot queries should be one major design goal for an efficient polyglot execution engine (see Section 4.3). In this experiment, we revisit the embedding approach to validate our assumption. To this end, we extend the Typer implementation of TPC-H Q6 and the C++ implementation of the Airline ETL query as representative workloads from Section 4.6.2 with state-of-the-art language runtimes. In particular, we use CPython [98]

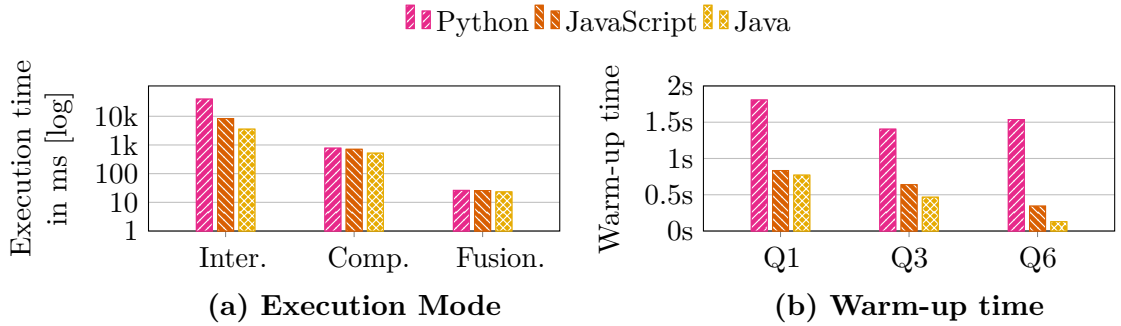


Figure 4.11: Impact of adaptive query compilation.

and PyPy [39, 262] for Python UDFs, V8 [114] and Duktape [272] for JavaScript UDFs, and HotSpot JNI [213] and GraalNative [209] for Java UDFs.

Results. In Figure 4.10, we show the overhead of all runtimes normalized to the execution time of the Typer/C++ implementation without UDFs.The experiment confirms our assumption that the embedding of language runtimes causes a substantial overhead of up to 112x in Python, 103x in JavaScript, and 58x in Java. When leveraging JIT compilation, PyPy and V8 improve performance by up to 5x compared to the interpreted counterpart. However, in comparison to the C++ baseline, they still cause an overhead of at least one order of magnitude. Furthermore, GraalNative compiles Java UDFs ahead-of-time into shared-libraries, which can be directly embedded in a data execution engine. This reduces the overhead of Java UDFs to 5x on average compared to the native Java implementation.

In contrast, Babelfish does not embed a foreign language runtime. Instead, Babelfish unifies built-in operators and UDFs in one execution engine and performs holistic optimizations across operator boundaries. As a result, Babelfish at least matches the performance of the hand-written baselines, independent of the UDF language.

Summary. Overall, this experiment validates our initial assumption. Across all language runtimes, we have observed a significant performance overhead. As a result, those runtimes cannot exploit the performance advantage of modern hardware. In contrast to embedding, Babelfish introduces a unified representation for polyglot queries. This enables Babelfish to optimize polyglot queries holistically, which results in a peak performance that is on par with hand-optimized, hand-written query implementations.

4.6.3.2 Just-in-Time Compilation

Babelfish leverages JIT compilation for query execution. First, it collects profiling information by interpretation and then leverages it during query compilation (see Section 4.4). In this experiment, we evaluate the impact of JIT compilation with respect to peak performance and warm-up time, i.e., the time it takes to reach peak performance after query submission. To this end, we evaluate different relational queries with varying complexity.

Results. In Figure 4.11a, we evaluate TPC-H Q6 that consists of a single pipeline with three compiler configurations: *1)* interpretation, *2)* compilation of individual operators, and *3)* compilation in combination with Babelfish's operator fusion. First, interpretation-based execution results in the highest query execution time. Second, the compilation of individual operators improves performance by up to 20x, as the code efficiency of individual operators

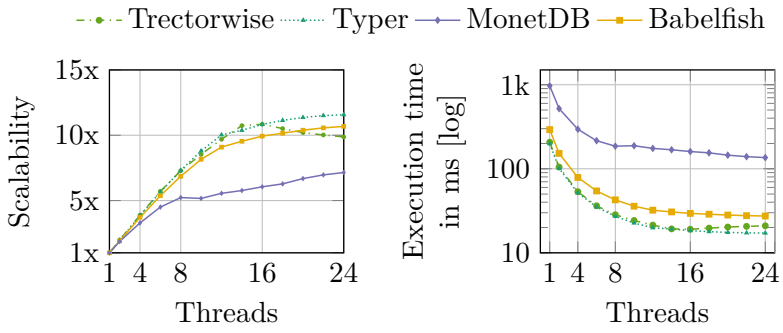


Figure 4.12: Scaling degree of parallelism on TPC-H Q6.

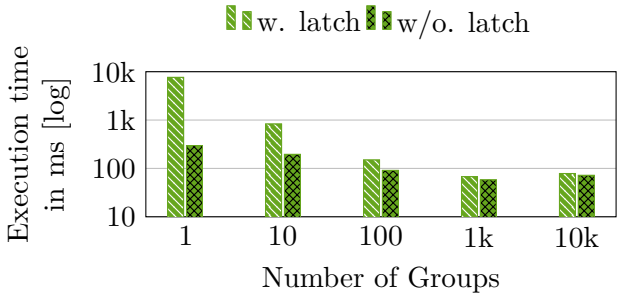


Figure 4.13: Contention handling for an global aggregation.

increases. Third, operator fusion further improves performance by up to 40x, as it eliminates function calls and intermediate objects. As a result, we see that holistic optimizations are crucial to reach high performance.

In Figure 4.11b, we show the warm-up time of Babelfish on three queries with different numbers of operators. Our observations are two-fold. First, the warm-up time highly depends on the UDF language. Thus, Java and JavaScript-based queries reach peak performance after 100ms to 800ms. In contrast, Python UDFs cause a very high warm-up time of several seconds. Second, the warm-up time depends on the query complexity and increases with the number of operators and predicates, e.g., Q1 and Q3. Overall, Babelfish's warm-up times are comparable to other Java-based query compilers like LB2 [259] and DBLAB [250]. However, we expect warm-up times to improve in the future as the Graal compiler becomes more mature.

Summary. In this experiment, we have shown that query compilation improves the performance of polyglot queries significantly. Furthermore, we have seen that Babelfish has on average a warm-up time below one second. This is in line with other high-level query compilation approaches [259]. Consequently, Babelfish is applicable for a wide range of UDF-based use-cases.

4.6.3.3 Scalability

In this experiment, we study two aspects of Babelfish's scalability in multi-core environments with stateful queries. First, we compare the performance of Babelfish, Typer, Tectorwise, and MonetDB on TPC-H Q6 with scale-factor 10 and different degrees of parallelism. Second, we study the impact of *latch-reduction* for stateful UDFs, as proposed in Section 4.5.4.1.

Results. In Figure 4.12, we evaluate the scalability across all systems with 1 to 24 execution threads. Overall, Babelfish achieves a similar speedup as Typer and Tectorwise

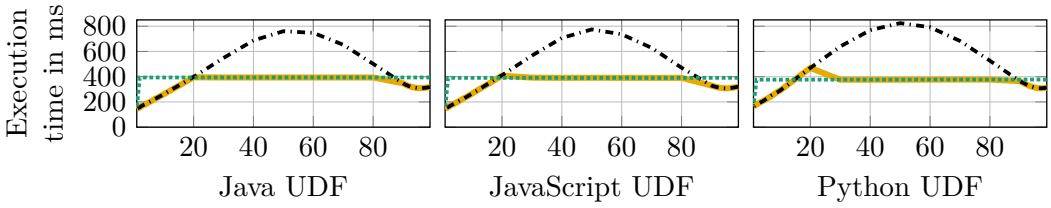


Figure 4.14: Impact of Predication for a selective UDF. Scaling selectivity from 0% - 100%.

(up to 10x). In contrast, the performance of MonetDB with 24 threads only improves by 5x compared to single-threaded execution. In comparison to Tectorwise and Typer, Babelfish causes only a slight performance degeneration of up to 30% and 60%. These results show that Babelfish is capable of utilizing multi-core environments efficiently.

In Figure 4.13, we study the performance impact of latch-reduction for stateful-UDFs with different levels of contention. To this end, we perform a grouped aggregation and increase the number of distinct keys. For a low number of keys (1-100), we see that the conversion of traditional latches to atomics using latch-reduction results in a significant performance advantage (up to 25x for single key aggregates). For increasing key ranges, contention decreases and the atomic version performs similar to a latch-based version. Overall, we see that latch-elimination mitigates the overhead in contention limited workloads.

Summary. In this experiment, we have identified two scalability characteristics. First, Babelfish scales with the degree of parallelism, outperforms MonetDB, and reaches a similar performance, compared to Typer and Tectorwise. Second, Babelfish's latch-reduction improves the performance of stateful UDFs in workloads with high contention. As a result, Babelfish efficiently executes UDF-based polyglot queries even in multi-core environments.

4.6.3.4 Predication

In this experiment, we assess the benefit of predication on a polyglot query that combines a relational aggregation and a selective UDF, which evaluates a filter on each input record. We conduct the experiment with filter selectivities ranging from 1% to 99% and Java, Python, or JavaScript UDFs. Furthermore, we evaluate three variants, i.e., branched, predicated, and adaptive. Branched always performs a conditional branch instruction. Predicated always applies predication and eliminates the branch. Adaptive leverages profiling information and applies branching or predication depending on the filter selectivity.

Results. Figure 4.14 below shows that branching and predication result in similar performance profiles across all UDF languages. In general, we observe that branching is beneficial for selective UDFs with low or very high selectivity. If the selectivity is between 5% and 80%, our results show that the branched variant suffers from branch-misprediction overhead. In contrast, the performance of the predicated variant is independent of the operator selectivity, which leads to a speedup of up to 2x for selectivities of 50% over the branched variant. Furthermore, the adaptive variant chooses between a branched or predicated execution based on filter selectivity. As Babelfish enables the collection of such profiling information during query interpretation, we can leverage this knowledge to select the best variant adaptively.

Summary. This micro experiment highlights the benefit of predication even for queries involving Java, Python, and JavaScript UDFs. This indicates that such traditional data processing optimization can yield a significant impact even on scripting languages. Furthermore,

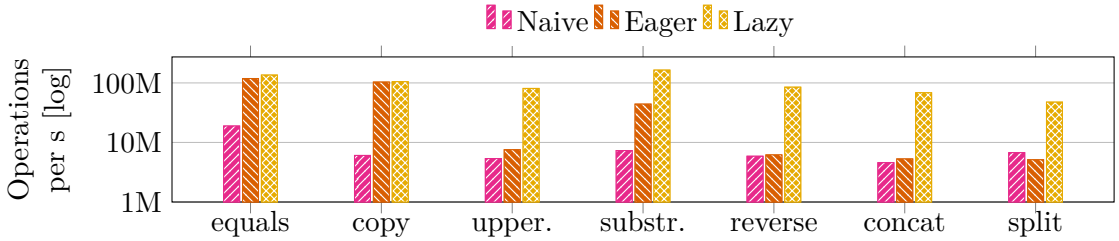


Figure 4.15: Performance of Babelfish's `naive`, `eager`, and `lazy` text processing strategies across common text functions.

our results are in line with recent work by Zhang et al. [1], who demonstrated the impact of predication across different JavaScript-based data processing queries.

4.6.3.5 Text Processing

Text manipulation causes a high performance overhead in current data processing systems, as shown in Figure 4.7b. To mitigate this bottleneck, Babelfish introduces `PolyglotRopes` to perform text processing lazily (see Section 4.5.4). In this experiment, we evaluate eight common text operations and compare the performance of three text processing approaches. *1)* `Naive` is used by common data processing systems and materializes text values as intermediate string objects before invoking UDFs. *2)* `Eager` passes text values as pointers to UDFs but performs all text manipulations eagerly, e.g., a `reverse` allocates an intermediate string object. *(3)* `Lazy` leverages Babelfish's `PolyglotRopes` and executes text manipulations lazily.

Results. In Figure 4.15, we observe that across all queries, `Naive` reaches the lowest performance, as it introduces additional string copies. For read-only text operations, i.e., `equals` and `copy`, `Eager` eliminates these allocations and operates directly on the raw pointers, which results in a 10x speedup. However, this benefit vanishes for text manipulations, e.g., `concat`, `reverse`, or `split`. For these manipulations, only `Lazy` is able to eliminate all intermediate string objects. As a result, `Lazy` outperforms the `Naive` approach by at least 10x across all text operations.

Summary. In this experiment, we have demonstrated that Babelfish's `PolyglotRopes` improve the performance of text operations by at least one order of magnitude. This observation explains the performance advantage of Babelfish across the data science workloads in Section 4.6.2. As a result, Babelfish is applicable for a wide range of data science tasks, which are usually very text-heavy.

4.6.3.6 Raw Data Processing

Polyglot workloads often process raw data [142], e.g., CSV or Arrow files. To enable high performance across different data formats, Babelfish interleaves data parsing and processing (see Section 4.5.4.2). To investigate the impact of this optimization, we execute TPC-H Q3 (three source relations) and Q6 (one source relations) across four data formats, i.e., CSV, Arrow, NSM, and DSM. For CSV and Arrow, we differentiate between an `Eager` (parses and de-serializes all data before processing) and `Lazy` (delays parsing if possible) execution mode. To exclude the overhead of I/O operations, we load all data to a memory before processing.

Results. In Figure 4.16, we make three observations. First, processing CSV data causes the highest execution time, as Babelfish scans the whole file to infer record boundaries and

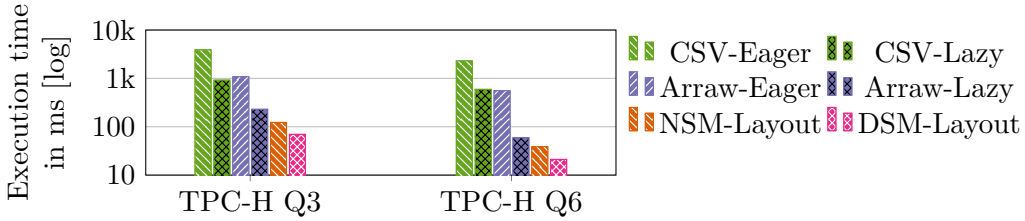


Figure 4.16: Performance of Babelfish across different data formats.

field accesses require costly de-serialization. Second, even though Arrow is an efficient memory format, it causes a 2x overhead compared to Babelfish, e.g., to convert Arrow data types to the Babelfish type system. Third, for CSV and Arrow, `Lazy` skips parsing, de-serialization, and materialization of unused fields. As a result, `Lazy` reduces query execution time significantly, i.e., up to 4x for CSV and 10x for Arrow.

Summary. This experiment has demonstrated that interleaving data parsing and processing significantly improves performance by up 4x on CSV and 10x on Arrow, independent of the number of source relations. Consequently, this optimization improves Babelfish's applicability for a wide range of use-cases.

4.6.4 Discussion

Our evaluation has shown that Babelfish accelerates the execution of polyglot queries significantly. In the majority of cases, Babelfish reaches the performance of hand-written C++ implementations, such as Typer and Tectorwise, without losing the generality of generic polyglot queries. Furthermore, we have demonstrated that Babelfish's performance optimizations, e.g., operator fusion, workload specialization, and latch-reduction, improve query performance significantly. For specific workloads, Babelfish even outperforms hand-optimized implementations as Babelfish is able to perform optimizations across operator boundaries.

4.7 Related Work

In this section, we contrast Babelfish to related work in the areas of *query compilation* and *polyglot queries* execution.

Query Compilation. Query compilation was extensively studied by Rao et al. [234], Krikellas et al. [166], and Neumann [202]. Over the last decade, it was applied in many data processing systems [6, 102, 166, 195, 202, 203, 215, 217, 278] and to different workloads, e.g., stream [31, 125, 147, 265, 289] and spatial data processing [260]. Babelfish leverages query compilation to execute polyglot queries efficiently. In contrast to many query compilation approaches, Babelfish does not directly generate intermediate code, e.g., Java, C++, or LLVM IR. Instead, operators in Babelfish are implemented in standard Java classes. At runtime Babelfish tightly integrates query processing with the Graal compiler to generate efficient machine code. Similar to LegoBase [161] and DBLAB [250], Babelfish defines the Babelfish-IR as a central representation of queries. Similar to LB2 [259], Babelfish leverages partial evaluation to specialize operators regarding query and data parameters. In contrast to these works, our Babelfish-IR captures operators beyond the relational algebra, e.g., UDFs, and

enables holistic optimizations. Further research focused on reducing the compilation latency by generating code for bytecode interpreters [162], custom compilers [103, 157], or mature JIT compilers [133]. Similar to this work, Babelfish leverages Graal as a JIT compiler that provides multiple compilation tiers to balance latency and throughput. Another line of research uses query compilation for the efficient processing of raw data formats, e. g. CSV or JSON [154, 155]. Babelfish applies this approach for the efficient execution of polyglot queries and fuses the parsing of raw data with the code of UDFs. Overall, Babelfish expands the application of query compilation to enable the optimization and efficient execution of polyglot queries.

Supporting Polyglot Queries. The efficient execution of polyglot queries and UDFs has been an active field of research [236]. In general, we can differentiate between three different approaches, i.e., the *translation* of UDFs to SQL statements, the introduction of *domain-specific languages*, or the *embedding* of polyglot runtimes in the execution engine. The first line of research focuses on the *translation* of complete UDFs [63, 64, 65, 84, 85, 88, 89, 233, 252] or particular 3rd-party libraries [119, 134, 149, 152] to equivalent relational expressions. With the integration of Froid [233] in Microsoft's SQL Server [212], this approach received a lot of attention. Froid converts loop-free UDFs into plain SQL queries. Based on this, Gupta et al. [132] proposed Aggify to optimize loops in UDFs. Duta et al. [84, 85] follow the same goal and convert PL/SQL UDFs to recursive common table expressions to support complex control flow, e.g., loops and recursions. In general, the translation of UDFs to SQL queries is a promising solution, as it eliminates UDFs. However, current approaches are limited to a subset of a particular UDF language or specific 3rd-party APIs. It is still unclear how complex language constructs could be supported, e.g., virtual function calls. Furthermore, translation can result in complex SQL queries, e.g., involving recursion, which could be hard to optimize [233]. In contrast, Babelfish embeds polyglot operators in different languages directly in the execution engine, supports all language constructs, and enables holistic optimization.

A second line of research proposed *domain specific languages* as UDF-based query languages [3, 14, 34, 110, 111, 112, 131, 136, 168]. They utilize an IR to enable advanced query optimizations, e.g., loop fusion and dead code elimination [3], selection pushdown [131], optimizations for distributed dataflows [14, 110, 111, 112], or the integration of different algebras [168]. However, DSLs limit users to a restricted set of programming constructs and operations. In contrast, Babelfish introduces optimizations for the efficient execution of polyglot queries with general-purpose UDFs in Java, JavaScript, and Python.

A third line of research focused on *embedding* polyglot operators directly in data processing systems [57, 76, 95, 108, 160, 170, 189, 201, 230, 251, 253]. These approaches mainly differ in the level of integration between the data processing system and the language runtime. Naive approaches pass tuples individually to the language runtime [189], which causes a high overhead for calling the external runtime. More advanced approaches leverage strided execution models to reduce this overhead [160, 230, 238]. In contrast, Babelfish embeds polyglot operators directly in the physical execution plan to eliminate these boundaries. Ishizaki et al. [145], Trill [56, 57], and Gerenuk [201] manipulate the source code of UDFs to remove object allocations and optimize memory access patterns. In contrast, Babelfish performs such optimizations independently of a UDF language on the Babelfish-IR. Schuele et al. [248], Sichert et al. [251], Tupleware [71], and Tuplex [253] translate UDFs to LLVM-IR

and fuse them with built-in operators. These approaches reach optimal performance but have two main drawbacks. First, LLVM-IR is a low-level assembly-like IR. This makes it hard to perform traditional database optimizations across operators, as it requires extracting data operator semantics from low-level IR [224]. Second, the translation of UDF languages, like Python, to LLVM-IR is a challenging problem by itself. Even a mature LLVM-based compiler like Numba [171] only supports a limited subset of Python. In contrast, our Babelfish-IR enables complex optimizations across operators and supports JavaScript, Python, and Java UDFs without restrictions. Additional work utilized Truffle to optimize data processing pipelines. In contrast to Babelfish, these approaches are limited to specific aspects of the overall data processing job, e.g., embedding R scripts [170], specializing CSV parsing [242], applying predication on JavaScript programs [297]. However, Babelfish is a complete execution engine for polyglot queries. In particular, Babelfish leverages Truffle and Graal as a foundation and proposes cross-operator optimizations to enable efficient execution. Similar to Babelfish, the recently proposed DynQ [243, 244] engine leverages Truffle to support polyglot queries but follows a dynamic type system.

4.8 Conclusion

In this chapter, we have presented Babelfish, a novel data processing engine optimized for polyglot workloads. Babelfish combines built-in operators and UDFs across different programming languages in one unified intermediate representation. This enables Babelfish to apply traditional database optimizations to the problem of polyglot queries and to specialize query processing to the specific requirements of UDFs. In particular, Babelfish performs holistic optimization across operator boundaries to eliminate the overhead of UDFs, e.g., operator fusion, predication, or workload specializations. As a result, Babelfish provides efficient execution of polyglot queries independent of the definition language of operators.

Our evaluation demonstrates that Babelfish outperforms traditional approaches for embedding polyglot operators by at least one order of magnitude and reaches the performance of hand-optimized implementations across various workloads. Thus, Babelfish bridges the performance gap between relational and polyglot queries and lays the foundation for the efficient execution of polyglot workloads. In particular, we introduce building blocks that can be integrated into various data processing systems to improve the support of UDFs. This will enable developers to implement data processing pipelines using their preferred programming languages without suffering from performance limitations. Consequently, our work makes data processing systems more accessible to a broader audience. To show the practicality of Babelfish, we leverage it for the acceleration of UDFs in NebulaStream.

4.8.1 Integration in NebulaStream

Using Babelfish, we integrated support for polyglot queries in NebulaStream. This enables the efficient execution of Python, Java, and JavaScript UDFs. In particular, Babelfish acts as a special-purpose accelerator that NebulaStream uses to offload specific operator pipelines. To provide this integration, we extended Babelfish in the following directions:

Holistic optimization of NebulaStream Operators and UDFs: We provided an implementation of NebulaStream intermediate representation (the Nautilus-IR) for Babelfish. This enables the holistic optimizations from Section 4.5, across build-in NebulaStream operators that are implemented in C++ and user-provided UDFs. Details are discussed in Chapter 5. Furthermore, we integrated NebulaStreams custom data types in Babelfish, e.g., variable size Text and Dates. As a result, UDFs can process data without any conversion overhead.

Isolating UDF execution: NebulaStream targets highly distributed IoT environments and provides efficient support for the execution of concurrent queries [60]. As a result, UDFs of different users must be isolated to prevent data corruption and leaks. To ensure isolation, we encapsulate Babelfish in lightweight virtual machines, using Cloud Hypervisor [140] and Firecracker [5]. To avoid an additional performance overhead NebulaStream fuses operators that require isolation and establish an efficient communication channel between the host and the isolated Babelfish runtime.

In summary, Babelfish allows NebulaStream users to design data processing pipelines in their preferred programming languages without compromising performance. Babelfish performs holistic optimization across all operators and avoids any overhead usually associated with UDFs in traditional systems. Moreover, Babelfish's architecture has inspired the creation of NebulaStream's novel query compilation framework, which we will discuss next.

5

Query Compilation Without Regrets

Cloud vendors, like Snowflake [74], Amazon [18] or Databricks [28], build high-performance query execution engines to elastically scale a variety of data processing workloads. The main engineering challenge for these engines is to balance performance and productivity. On the one hand, an engine has to provide high-performance query execution for a wide range of workloads from various end users [28]. To achieve this, system engineers have to develop efficient data processing operators, which involve traditional relation operators as well as specialized operators for stream processing, machine learning, or user-defined functions (UDFs) [128]. On the other hand, the engine has to ensure a high productivity for engineers to enable the timely integration of new features. To this end, the engine has to be easy to modify, test, and debug [28]. As a result, engineers have to choose a suitable software architecture that aligns with their specific requirements.

5.1 Introduction

Over the last decade, *vectorized query interpretation* [40] and *query compilation* [202] have emerged as state-of-the-art architectures for high-performance query execution engines. Vectorized query interpretation extends the traditional Volcano processing model [117] and passes chunks of records between precompiled operators that can be developed in imperative code. In contrast, *query compilation* translates queries into specialized machine code that is compiled at runtime. This allows the data processing engine to reach high execution performance at the cost of a compilation-time overhead. To compile data processing queries, research has proposed different specialized query compilation strategies. These approaches either optimize for short-running queries [103, 157], target specific hardware [225], propose performance optimizations [72, 194], or focus on particular workloads [70, 125, 128]. However, even though query compilers enable high performance, their development and maintenance is complex. Thus, data processing systems struggle with their integration [28, 122].

In particular, we identify two challenges that system engineers face while designing state-of-the-art query compilers. First, the design space of a query compiler is very large, and there is no approach that is optimal for all workloads [121]. Thus, engineers must build

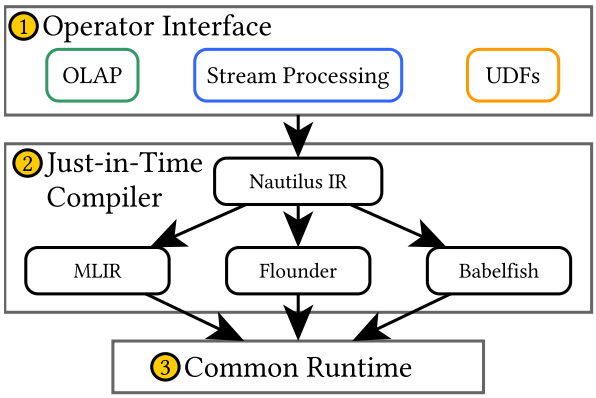


Figure 5.1: Overview of the Nautilus framework.

workload-specialized query compilers that balance compilation time, execution performance, and engineering effort [157]. Second, query compilers generate the implementation of operators only after query submission, i.e., at runtime. This introduces an indirection between the implementation and the execution of operators, which makes compilation-based engines hard to build and debug [28, 218]. This is particularly problematic for query compilers that use specialized compiler frameworks like LLVM as they require engineers to have a deep understanding of compiler technology [161]. This makes it difficult to hire and onboard qualified engineers [198, 218]. As a result of both challenges, recent commercial data management systems like Photon [28] and FireBolt [218] have adopted vectorized query interpretation in contrast to query compilation to ensure high productivity of their engineers.

To address these challenges, we propose a novel way for systems to benefit from the advantages of query compilation without sacrificing productivity. In particular, we propose Nautilus, a framework for developing data processing engines that bridges the gap between query interpretation and query compilation. To this end, Nautilus combines an interpretation-based programming model to ensure high productivity with a novel trace-based just-in-time (JIT) compiler, which translates operators into efficient code. Figure 5.1 shows the three main components of Nautilus. First, Nautilus provides a generic interface for implementing diverse data processing operators ①. This avoids the complexity of code generation and enables engineers to write imperative C++ code that is easy to develop, debug, and maintain. Second, Nautilus's JIT compiler traces imperative operators to derive a unified intermediate representation, the Nautilus IR, and provides multiple execution backends to produce efficient code ②. This enables Nautilus to specialize query execution towards particular workload requirements, e.g., minimizing startup latency or maximizing execution performance. Third, Nautilus introduces a common interface between the executable operators and the host runtime that can be used across all execution backends ③. This simplifies the implementation of operators and enables engineers to reuse and implement common data structures, e.g., hash-tables, lists, and additional data types. To demonstrate the effectiveness of Nautilus in practice, we use it as a query compiler of our data processing platform NebulaStream [289]. Our evaluation shows that Nautilus achieves high performance for short-running, long-running, and UDF-based data processing workloads. As a result, Nautilus combines the ease of use and productivity of query interpretation with the flexibility and performance of state-of-the-art query compilers. Thus Nautilus enables query compilation without regrets.

In summary, our contributions are as follows:

1. We introduce Nautilus to unify the ease of use of query interpretation with the performance of query compilation.
2. We present a novel operator implementation interface that is easy to maintain, extend, and debug.
3. We propose a novel trace-based query compiler to translate imperative operators into efficient machine code on different backends.
4. We integrate Nautilus into our new data processing system NebulaStream and demonstrate Nautilus's high performance across diverse workloads.

The rest of this chapter is structured as follows: First, we discuss the challenges of query compilation in execution engines (see Section 5.2). Based on this, we introduce Nautilus (see Section 5.3), our operator implementation interface (see Section 5.4), and our novel trace-based JIT-compiler (see Section 5.5). Then, we evaluate Nautilus across different workloads (see Section 5.6). Finally, we discuss related work (see Section 5.7) and conclude (see Section 5.8).

5.2 The Curse of Query Compilation

Over the last decade, query compilation has been applied in many data processing systems to maximize execution performance [17, 18, 195, 202]. Even though query compilation is widely adopted, it introduces a high system complexity. The engineers at Databricks recently discussed the engineering challenges of their query compiler for SparkSQL [28]. They argue that with a vectorized interpretation-based architecture, it is easier to develop and scale the engine. Due to this benefit, commercial systems such as Photon [28], FireBolt [218], and Velox [221] follow interpretation-based architectures to reduce development costs and ensure the high productivity of their engineers. Thus, the consensus has emerged that (i) compilation-based engines reach superior execution performance, but (ii) interpretation-based engines are much easier to build and maintain. Consequently, getting the best of both worlds remains a desirable goal. In general, we identify two challenges that hinder the adoption of query compilation.

Challenges 1: Managing high engineering effort. As any other software artifact, building, debugging, and maintaining query execution engines requires developer time and costs. Consequently, it is necessary to maximize productivity of engineers to reduce development costs. In contrast to interpretation-based engines, query compilers generate operator code after query submission, i.e., at runtime. This introduces a gap between implementation and execution, which makes the engines hard to build, debug, and maintain. For instance, Databricks reported that developing a query compiler caused 8× higher engineering costs compared to an interpretation-based engine [28]. Furthermore, state-of-the-art query compilers use code generation frameworks such as LLVM [202] or build custom compilers [103, 157]. These compilers reduce the compilation time to support short-running queries but require a deep understanding of specialized compiler frameworks. In particular, they generate code which resembles assembly and is highly complex. As a result, it becomes challenging to find engineers that have expertise in compiler technology and data management [218]. This is particularly problematic for academic projects like NebulaStream, as many students struggle with system engineering [198]. Even for experienced engineers, debugging the generated code

is cumbersome [278]. To overcome this challenge, a compilation-based engine has to hide code generation details and focus on high productivity.

Challenges 2: Navigating a large design space. Modern data processing systems support an increasingly diverse set of workloads and hardware. For example, NebulaStream targets streaming, batch, and ML workloads in heterogeneous IoT environments. To this end, research introduced specialized query compilers, e.g., for short-running queries [157], stream processing [125, 265], user-defined functions [70, 128], and heterogeneous hardware [225]. These compilers introduce different architectures that make different trade-offs between compilation time, execution performance, and developer experience. However, there is currently no query compiler available that is suitable for a wide range of workloads. As a result, engineers must develop compilation-based engines from scratch and reinvent solutions for reoccurring tasks like the integration of compiler frameworks [151]. Furthermore, system engineers must develop and maintain different query compilation backends to efficiently support diverse workloads. For instance, Umbra has two compilation backends to support short- and long-running queries [157]. To navigate this versatile design space, a compilation-based engine should offer a framework to integrate different execution strategies that optimize for specific workloads.

Overall, both challenges increase the engineering effort and the cost of compilation-based query execution engines. Besides commercial vendors, this also impacts research as most groups cannot afford such engineering efforts. Therefore, we propose to commoditize query compilation technology for a broad user base. To this end, we propose Nautilus, our query compilation framework that bridges the gap between query interpretation and compilation.

Opportunities: Addressing these challenges opens a wide range of new opportunities for industry and research. In particular, Nautilus could enable industry to leverage query compilation for the acceleration of specific workloads and researchers to focus on specific research contributions. For instance, data processing systems such as NebulaStream can leverage Nautilus to avoid implementing a query compiler infrastructure from scratch. Furthermore, we see Nautilus supplementary to the currently evolving field of composable data processing systems [75, 151, 221]. For instance, Velox [221] could leverage Nautilus to introduce query compilation for specific operators. Furthermore, Nautilus could integrate Daphne [75] or LingoDB [151] as compilation targets and benefit from their domain-specific optimizations. In general, we envision Nautilus as the first step towards accessible query compiler technology.

5.3 Nautilus: A Query Compilation Framework

In this section, we present Nautilus, our novel and extensible framework for implementing compilation-based query execution engines. Nautilus addresses the previously outlined challenges and enables engineers to utilize query compilation without compromising on productivity. In particular, Nautilus decouples the implementation of operators from the actual query execution. It achieves this through three primary aspects: First, it provides an interface for the implementation of imperative data processing operators that focuses on developer productivity. Second, it incorporates a JIT compiler that specializes query execution towards specific workload requirements. Third, it is system-agnostic to enable integration

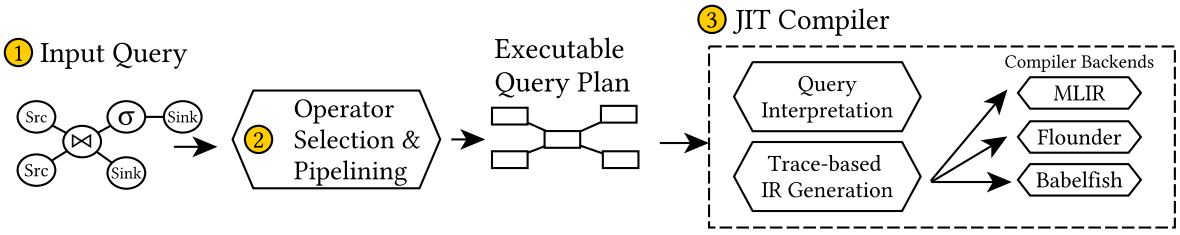


Figure 5.2: Overview of query execution in Nautilus.

in different host systems. This versatility allows engineers to employ Nautilus either as a foundation for an execution engine or as a specialized accelerator for distinct workloads.

In the remainder of this section, we first provide an overview of Nautilus's query processing model (see Section 5.3.1) and discuss its extensibility (see Section 5.3.2). Subsequently, we introduce Nautilus's operator implementation interface (see Section 5.4) and its JIT compiler in detail (see Section 5.5).

5.3.1 Query Execution in Nautilus

In this section, we provide an overview of Nautilus's query processing phases (illustrated in Figure 5.2) and discuss its main components. To this end, we assume that Nautilus is embedded in a host query execution engine of a system like NebulaStream, which targets a variety of diverse data processing workloads, e. g. relational data processing, stream processing, and UDFs. In the following, we discuss how Nautilus represents queries ①, how it creates executable query plans ②, and how it executes them via its novel multi-backend JIT compiler ③.

Input Query: In the first phase ①, Nautilus receives an already optimized query plan from the host system. Query plans can access multiple sources, emit data to different sinks, and combine relational operators, ML operators, or UDF-based operators. This allows Nautilus to support queries for a wide range of data processing workloads **(C2)**.

Pipelining & Executable Operators: In the second phase ②, Nautilus selects for each operator a corresponding executable operator that provides the operator implementation. To this end, Nautilus's operator implementation interface enables engineers to implement operators using imperative code without any knowledge of query compilation. Based on this, Nautilus segments the query plan into pipelines [202]. Each pipeline scans a batch of data, invokes a sequence of operators, and terminates with an operator that materializes results, i.e., a natural pipeline breaker like a join build [202]. Depending on the number of operators in a query, the space of all possible execution plans becomes very large [129]. To navigate this space, Nautilus provides extension points for engineers to specialize the operator selection and segmentation [70, 72, 129, 194].

Query Execution: In the final phase ③, Nautilus uses JIT compilation to execute operator pipelines. Nautilus's JIT compiler supports both interpretation and compilation-based query execution. The interpreter directly executes the implementation of operators. This makes them easy to debug without the indirection of code generation (**C1**) but also limits the execution performance. In contrast, Nautilus's compiler translates all operators within a pipeline to specialized machine code (**C2**). To this end, Nautilus first traces the query and translates all

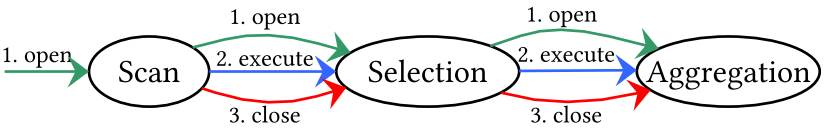


Figure 5.3: Executio n of operators within a pipeline.

operators to its Nautilus IR. Nautilus IR decouples the implementation of operators from the code generation and simplifies the implementation of compilation backends. Nautilus's JIT compiler provides several compilation backends to specialize code generation for particular workloads (**C2**), e. g. it offers high-performance backends to maximize the performance of long-running queries, low-latency backends to support short-running queries, and specialized backends to accelerate workloads like UDFs. Across all execution backends, Nautilus provides a common runtime to interact with the host data processing system.

5.3.2 Extensibility

Over the lifecycle of a data processing system, the target workloads of the system may expand. As a result, system engineers have to adjust and extend the functionality of the data processing engine (**C2**). To this end, they can extend Nautilus via *plugins* and *runtime functions*.

Plugins enable system engineers to extend Nautilus with new functionality on three levels. First, plugins can provide new strategies for the instantiation of executable operators and query segmentation, for instance, to use hybrid pipelining strategies [70, 72, 194]. Second, plugins can define new operators, expressions, or data types. To this end, Nautilus provides handlers to intercept logical and arithmetical operations during execution. This allows a plugin for spatial data types to reuse existing expressions and operators. Third, plugins can extend Nautilus's JIT compiler and provide new compiler backends to accelerate specific workloads.

Runtime Functions provide a bridge between the implementation of operators and the runtime system of the host engine. To this end, they expose pre-defined functions from the runtime to the implementation of an operator and transparently convert intermediate values. This enables system engineers to decouple common functionality from concrete operators, e.g., a join may call into a pre-compiled hash table implementation. At runtime, proxy functions either result in direct function calls or are automatically inlined in the actual pipeline.

In summary, extensibility enables engineers to extend Nautilus-based execution engines in several ways. This improves the flexibility and maintainability of the resulting engine.

5.4 The Operator Implementation Interface

Developing operators for query-compilation-based data processing engines is complex [28]. In particular, the indirection of generating operator implementations at runtime makes development challenging (**C1**) and hinders the support for diverse workloads (**C2**). To address these challenges, Nautilus provides an easy-to-use interface for operator implementations that aims for three design goals: First, it follows an interpretation-based processing model that decouples individual operators. Second, it allows engineers to implement operators in generic imperative C++ code using lightweight abstractions that are easy to test and debug. Third, it

Listing 5 Scan operator.

```
1 class Scan : public Operator {
2  void open(RuntimeCtx& ctx, TupleBuffer& tb){
3   // calls open on all child operators
4   child->open(ctx, tb);
5   // iterates over tuples in buffer tb
6   auto nrTuples = tb.getNumTuples();
7   for (Value<UInt64> i = 0; i < nrTuples; i++){
8    // reads a record from the tuple buffer and
9    // passes it to the child operator
10   auto tuple = tb.read(i);
11   child->execute(ctx, tuple);
12  }
13 }}
```

Listing 6 Selection operator.

```
1 class Selection : public ExecutableOperator{
2  void execute (RuntimeCtx& ctx, Tuple& tuple){
3   // calls child operator if expression is true
4   if (expression->execute(tuple))
5    child->execute(ctx, tuple);
6 }};
7
8 class LessThanExpression : public Expression{
9  Value execute(Tuple& tuple){
10  auto leftValue = leftSubExp->execute(tuple);
11  auto rightValue = rightSubExp->execute(tuple);
12  return leftValue < rightValue;
13 }};
```

provides abstractions for common data types and allows the integration with the host system. Using this interface, engineers can implement operators in a maintainable way, while it enables Nautilus to generate efficient code at runtime. In the remainder of this section, we discuss individual aspects of this interface in detail and use the operator implementation of TPC-H Q6 as a running example. This query first scans the input data (see Listing 5), performs a set of selections (see Listing 6), and finally aggregates an expression (see Listing 8).

5.4.1 Pipeline Evaluation

During query execution, pipelines receive data from predecessor pipelines and emit intermediate results to successor pipelines. Within pipelines, Nautilus pushes data from one operator to another via function calls (illustrated in Figure 5.3). Thus, Nautilus follows a push-based execution model [202]. In contrast to the pull model of traditional interpretation-based engines [117], the push model aligns the control- and data-flow between operators. Both follow the same direction, i.e., from the initial `Scan` towards the most downstream operator. This simplifies the implementation and debugging of individual operators (**C1**).

Similar to the Volcano processing model [117], Nautilus defines a simple interface for implementing operators. Operators may implement, `setup()`, `teardown()`, `open()`, and `close()` to specify their processing logic. `Setup()` and `teardown()` are called once per operator and initialize/clear global operator state, e.g., the hash-table in a grouped aggregation. `Open()` and `close()` are invoked for each buffer of data and allow operators to maintain local state, for instance an emit operator allocates an output buffer to materialize results. Furthermore, children of the scan operator implement the `execute()` function to process individual tuples. For example, the selection in Listing 6 evaluates an expression on each input tuple. Additionally, operators receive a reference to the `RuntimeContext` that maintains operator state. As a result, Nautilus's push-based processing model and operator interface decouples operators, simplifies their implementation, and improves testability (**C1**).

5.4.2 Imperative Operator Implementation

For the implementation of individual operators, Nautilus provides a lightweight C++ interface that focuses on simplicity and expressiveness (**C1**). Operators can be entirely implemented in generic, imperative C++ code. Thus, engineers are relieved from the complexity of code generation and can express data processing logic using common data types, function calls, and

Listing 7 Aggregation operator.

```
1 class Aggregation : public ExecutableOperator{
2  void execute(RuntimeContext& ctx, Tuple& tuple){
3   auto state = (AggStates*) ctx.operatorState(this);
4   // executes all aggregation functions
5   for (auto i = 0; i < aggregations.size(); i++){
6    auto value = aggregations[i].lift(tuple);
7    aggregations[i]->lift(state->agg[i], value);
8   }
9  }
10 };
```

Listing 8 Sum Aggregation Function.

```
1 class SumFunction : public Aggregation{
2  Value lift(Tuple& tuple){
3   return inputExpression->execute(tuple);
4  }
5  void update(State* state, Value& value){
6   state->currentSum += value;
7  }
8  Value lower(State* state){
9   return state->currentSum;
10 }};
```

control-flow statements, e.g., `if`, `for`, or `while`. For example, the `Scan` operator uses a simple `for` loop that iterates over the content of a data buffer (see Line 7 in Listing 5). In particular, Nautilus defines three abstractions to process data, i. e., `TupleBuffers`, `Tuples`, and `Values`.

TupleBuffers represent chunks of memory that store data according to a specific data layout and provide methods to read and write tuples at particular positions.

Tuples represent individual data entries as record types and define a collection of field names and associated values.

Values represent data elements of a particular type and can be part of a `Tuple` or the result of an operation, e.g., the evaluation of the `LessThanExpression` in Listing 6. As `Values` contain a concrete data element, system engineers can directly inspect their content at runtime. `Values` can be either primitive types, e.g., `Int8`, `Double`, `Ptr`, collection types, e.g., `Array<Int64>`, or composed types, e.g., `Point`. Furthermore, Nautilus uses operator overloading to provide logical and arithmetical operations between `Values`. For example, the `LessThanExpression` in Listing 6 Line 12 evaluates < on its inputs.

This abstraction decouples the operator implementation from the physical data representation. This allows engineers to support different data layouts, e.g., NSM, DSM, or PAX, without adjusting the implementation of operators. Furthermore, it enables engineers to split complex operators in individual and isolated components, i.e., sub-operators [163]. One example is the Aggregation operator in Listing 7. It maintains a global state in the `RuntimeContext` and passes each tuple to a set of aggregation functions in Line 6. All aggregation functions implement the same generic interface as proposed by Tangwongsan et al. [261], and provide three functions `lift()`, `combine()` and `lower()` see Listing 8. Lift transforms a tuple to a partial aggregate. Combine computes the combined aggregate from partial aggregates and updates the current aggregation state. Lower transforms a partial aggregate to a final aggregate. This allows for the reuse of aggregation function implementation across various physical operators, such as keyed and global aggregation for batch data and window aggregations in stream processing.

5.4.3 Data Structures

So far, we have discussed the implementation of individual operators, how they encapsulate processing steps, and operate on data items. Additionally, Nautilus provides common data structures, like `Lists` and `HashTables`, which can be used across different physical operators.

One key aspect in the design of these data structures is the boundary between operators and the runtime of the host system. To this end, query compilers typically employ one of two

Listing 9 Hash Join Prob.

```
1 class JoinProbe : public ExecutableOperator{
2  void execute(RuntimeContext& ctx, Tuple& tuple){
3  // derive key values
4  std::vector<Value<>> keys;
5  for (const auto& exp : keyExpressions) {
6   keys.emplace_back(exp->execute(record));
7  }
8  // calculate hash
9  auto hash = hashFunction->calculate(keyValues);
10 // load the reference to the global hash map.
11 auto hashMap = (HashMap*)ctx.getOperatorState(this);
12 // lookup the key in the hashmap
13 auto entry = hashMap.findOne(hash, keyValues);
14 // check if join partner was found
15 if (entry != nullptr) {
16  // Create result record load values from
17  // the probe side and store them in result.
18  for (size_t i = 0; i < entryFields.size(); i++) {
19   record.write(entryFields[i], entry[i]);
20  }
21  this->child->execute(ctx, record);
22 }
23 }
```

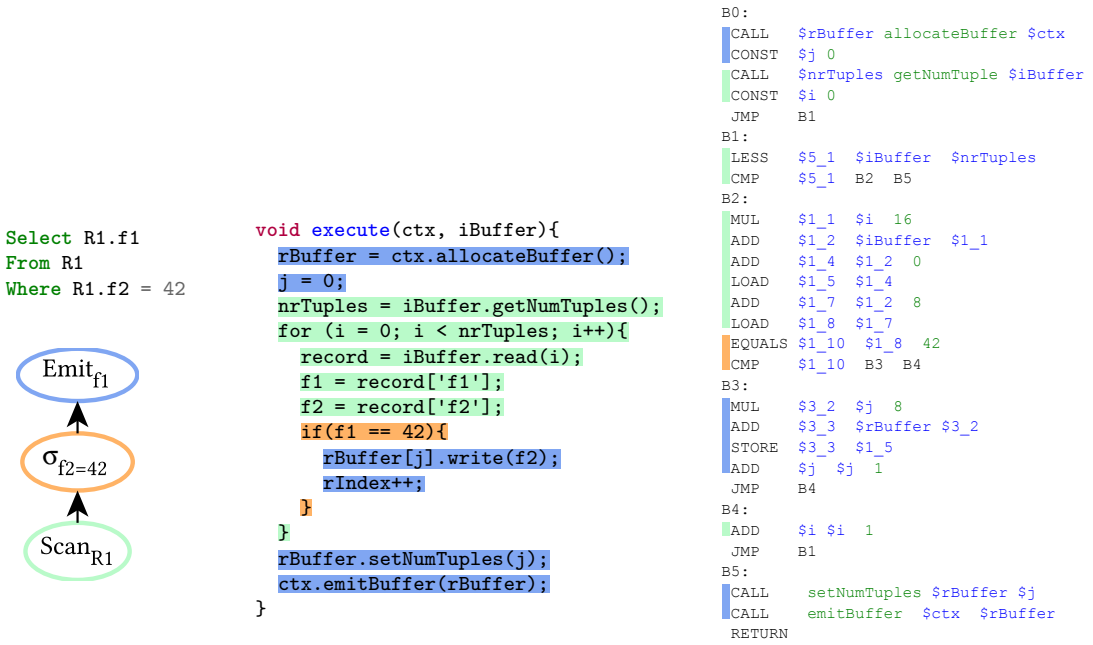
Listing 10 Hash Map Interface.

```
1 class ChainedHashMap {
2  Entry findOne(Value<UInt64> hash,
3                vector<Value> keys){
4   // call runtime function to find chain
5   auto e = FuctionCall<>(findChain, hash);
6   // iterate chain and search for entry
7   for (; e != nullptr; e = e.next){
8    if (compareKeys(e, keys)) {
9      break;
10   }
11  }
12  return entry;
13 }
14 Entry findOrCreate(Value<UInt64> hash,
15                    vector<Value> keys,
16                    function<> onInsert){}
17 ...
18 };
```

methods, as discussed in Section 2.2.4.2. The first method links the generated operator code to pre-compiled data structures in runtime libraries [202]. This provides a clear separation of concerns but introduces additional function calls between the generated code and the runtime, potentially impacting performance. The second method generates code for both operators and data structures [133, 161]. This improves code efficiency but requires the re-implementation of all data structures within the code generation framework, which decreases maintainability.

In Nautilus, we support both methods, allowing engineers to balance performance optimization and engineering effort. To realize this, Nautilus offers C++ data structure wrappers, such as vectors and lists as well as specially optimized data structures. For instance, the `HashJoinProb` operator in Listing 9 accesses entries in the `Hashmap` from Listing 10 to identify join partners. The operator selects key values (Lines 4-7) for each input tuple and calculates the hash (Line 9). Subsequently, it invokes `findOne(hash, keyValues)` on the hash map data structure. Listing 10 shows the implementation of this function using a simple chained hash map. Initially, it carries out a function call in the runtime to locate the chain corresponding to the specific hash value. Then, it iterates through all entries in the chain, comparing their keys. If a matching entry is found, it returns to the `HashJoinProb` operator. At runtime, Nautilus generates a single code fragment for both the `HashJoinProb` operator and the `findOne` function.

Since the `Hashmap` is defined through a generic interface, its concrete implementation is decoupled from individual operators. This flexibility empowers engineers to employ diverse implementations tailored to specific workloads, such as choosing between a chained or linear hash map. Moreover, it simplifies the testing and debugging processes for these data structures.



(a) Example Query **(b) Pseudocode of fused pipeline** **(c) Trace of pipeline**

Figure 5.4: Illustation of intermediate trace **(c)** for an example query **(a)**. The query performs a `scan`, `selection`, and an `emit` operator, which execute the set of operations on `Value` objects illustrated in the pseudocode of the fused pipeline **(b)**. The resulting IR is shown in Figure 5.5.

5.5 Trace-based Just-in-Time Compilation

In contrast to traditional query compilers [202], Nautilus's executable operators and data structures do not implement code generation. Instead, they are implemented in generic C++ code to ensure a good developer experience (**C1**). To translate these operators to efficient code, Nautilus introduces trace-based JIT compilation for data processing queries.

A tracing JIT compiler dynamically optimizes hot code paths during the execution of a program [23]. To this end, it first executes a program in an interpreter and records all executed instructions (the trace). If the same trace was executed multiple times, e. g. because is is part of a loop, the compiler translates the trace to machine code. In this case, the generated code only covers the hot code path of the original program. Today, this technique is the foundation of JIT compilers like PyPy [39] for Python and TraceMonkey [106] for JavaScript.

In contrast to these compilers, Nautilus operates on operator pipelines, which always contain a scan over some data and a set of operators that process individual records. This eliminates the need for the initial interpretation to detect hot-code paths. Instead, Nautilus uses symbolic execution [43] to trace all possible execution paths through an operator pipeline. In particular, it translates pipelines to efficient code in three steps: *1)* Nautilus uses a linear algorithm to record a trace of all the operations a pipeline of operators executes. *2)* Nautilus translates the trace to its intermediate representation, i.e., the Nautilus IR. *3)* Nautilus generates efficient machine code using specialized compilation backends. This enables Nautilus to balance compilation time and execution performance (**C2**).

In the following, we discuss tracing, Nautilus IR, and the compilation backends in detail. To this end, Figure 5.4 illustrates a running example of the individual steps of Nautilus's JIT

Algorithm 1 Trace Pipeline

```
1: visited tags Θ ← {}
2: execution paths E ← {[]}
3: while E ≠ ∅ do
4:     ϵ ← dequeue(E)
5:     Execute pipeline with ϵ
6: end while
```

Algorithm 2 Trace Operation

```
1: if op_tag ∈ ϵ then
2:     op executed in same execution → handle loop
3: else if op cause control-flow split then
4:     if op was executed the first time (op_tag ∉ Θ) then
5:         append(E, ϵ)
6:         returnValue ← true
7:     else if op was executed the second time then
8:         returnValue ← false
9:     else
10:        terminate execution of ϵ
11:    end if
12: end if
13: append(ϵ, op)
14: append(Θ, op)
```

compiler. From the initial query (a) Nautilus creates an executable query plan that fuses individual operators to data-centric pipelines (b). During JIT compilation, Nautilus executes the pipeline symbolically, creates the trace (c), and converts it to a Nautilus IR fragment.

5.5.1 Tracing Data-Processing Queries

In order to enable the tracing of data processing queries, Nautilus follows three key observations: *1)* Operators within a query are constant and do not change during execution. *2)* The targets of function calls between operators and the host runtime are constant. *3)* Operations that do not depend on any input data, represented by `Value` objects, can not change during query execution and are constant. This allows us to differentiate between operations that are *runtime constant* and *runtime dynamic* during tracing. For example, the `scan` in Figure 5.4 accesses a constant number of fields (`f1` and `f2`) for each tuple, whereas the actual values of the individual attributes are only determined at runtime, i. e., they are runtime dynamic.

This enables Nautilus to efficiently trace the implementation of operators. During tracing, Nautilus follows Algorithm 1 and 2. They executes pipelines *symbolically* to record all runtime dynamic operations that involve `Value` objects in a lightweight trace object.

Symbolic tracing. During symbolic execution, Nautilus executes pipelines multiple times using dummy data. In each execution, it intercepts all operations that involve `Value` object and appends them to the trace, e.g., the expression `f1 == 42` results in an `EQUALS` instruction in the trace. These instructions capture references to input and result `Values` as well as a unique `tag` to identify if the same operation was executed multiple times, e.g., as part of a loop in the `scan` operator. As the trace captures only the operations and their dependencies, the actual data values do not impact the trace and are not recorded.

A critical requirement for the trace is that it captures all potential execution paths during query evaluation. For instance, in our running example, the trace has to contain both outcomes and the resulting control flow of the `if` statement in the `selection` operator. To this end, Algorithm 1 evaluates the pipeline till all execution paths have been visited. It maintains a set of operation `tags` Θ and a queue of in-flight execution paths E. Each execution path $\epsilon \in E$

captures a unique sequence of operations that have been executed during query evaluation. As long as execution paths are inflight, Nautilus evaluates the pipeline again. For each traced operation, Algorithm 2 checks if the operation is part of a loop or if it causes a control-flow split, e.g., by a `if` statements. Control-flow splits require another pipeline evaluation such that Nautilus can also trace the other control-flow branch. To this end, Nautilus checks if the operation was executed before ($op_{tag} \in \Theta$). If it is executed for the first time, it appends the current execution path ϵ to the set of in-flight executions E and continues evaluating the `true` case. If it visits the split for the second time, it evaluates the `false` case in contrast. If it revisits the same control-flow split, Nautilus terminates the pipeline evaluation as both control-flow sides are already part of the trace. As a result, the tracing algorithm requires $\mathcal{O}(2n)$ iterations, respectively pipeline evaluations, in the worst case to evaluate all execution paths for a pipeline with n control-flow splits. Thus, even a complex pipeline with multiple nested operators requires only a small number of iterations.

In our running example, Nautilus traces the operation of the `scan`, `select`, `emit` operators as part of the same pipeline. As the function calls between operators are constant, tracing automatically fuses the operators, which results in the code illustrated in Figure 5.4 b. For each executed operation, Nautilus adds an instruction to the trace (see Figure 5.4 c). Both the `scan` and `select` operator introduce a control-flow split. In the first iteration, Nautilus enters the loop body of the `scan`, evaluates the `true` case of the `selection`, and leafs the loop as it executes the loop header (`i < nrTuples`) for the second time. Consequently, Nautilus only requires a second iteration to evaluate the `false` case of the `selection` and can terminate tracing early. The resulting trace covers all visited execution paths through the query and represents control flow via branches and basic blocks.

5.5.2 Nautilus IR

Nautilus introduces the Nautilus IR as a unified intermediate representation to decouple the implementation of operators from a specific compilation backend. In general, Nautilus IR follows three design goals to simplify the implementation of compilation backends: *1)* The IR is agnostic to specific operators and compilation backends. *2)* The IR focuses on a small set of operations and data types. *3)* The IR supports transformations to enable backend-specific optimizations. To this end, Nautilus IR balances compactness and expressiveness.

Nautilus IR follows static single-assignment (SSA) form and differentiates between functions, basic blocks, and operations (see IR in Figure 5.5). Functions usually correspond to operator pipelines and contain a sequence of basic blocks representing the control flow (illustrated with red arrows). Each basic block receives block arguments, defines a sequence of dataflow operations, and terminates with a control-flow operation. Operations may depend on input operations and produce at most one result value of a primitive type (illustrated with blue arrows). Nautilus IR provides common data flow operations, i.e., logical and arithmetical expressions, function calls, load, and stores, as well as control-flow operations for jumps and if conditions. To pass values between basic blocks, each block can define a set of block arguments [175]. In contrast to traditional SSA ϕ nodes, this simplifies tracking dependencies between values and operations.

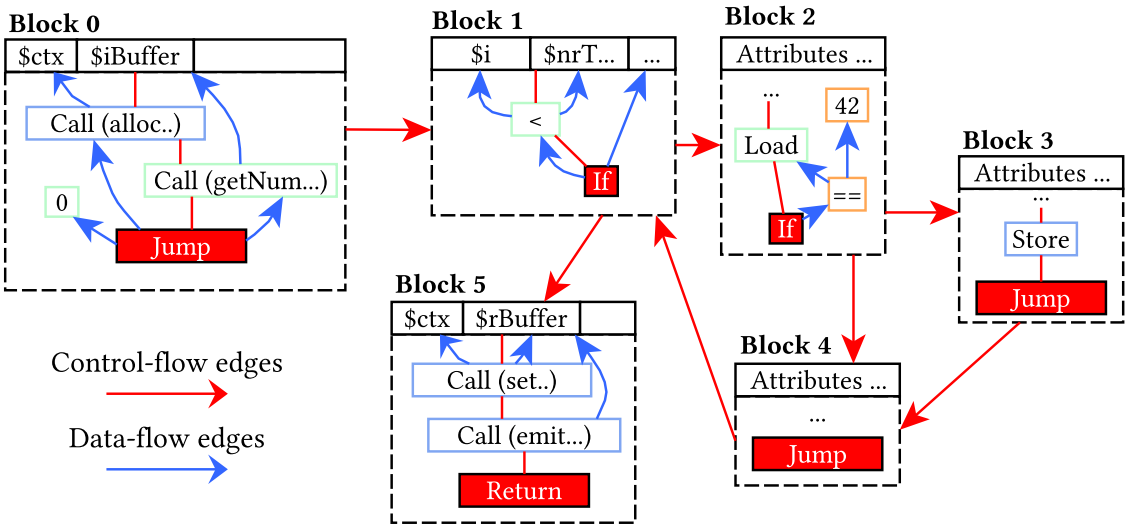


Figure 5.5: Illustration of the Nautilus IR for the example query and trace from Figure 5.4.

The IR for our running example results in six basic blocks (see Figure 5.5). *Block* 0 initializes local variables for the `scan` and `emit` operators. *Block* 2 and *Block* 4 are part of the `scan` and contain the loop head (`i < nrTuples`) and latch (`i++`), which has a backedge to the loop head. *Block* 3 loads the record fields (`f1`, `f2`), and evaluated the `selection`. If the predicate is `true`, the execution invokes *Block* 3 and stores `f2`. If the `scan` terminates, the loop header invokes *Block* 5, which is the loop exit block and `emits` the result buffer.

In general, Nautilus IR represents an intermediate step between the operator implementation and a specific compiler backend. This enables Nautilus to generalize optimizations across different compiler backends, i.e., to detect loop patterns or to perform constant-folding.

5.5.3 Compilation Backends

As discussed in Section 2.2.3, research proposed a variety of query compilation approaches that use different intermediate representations and target different workloads, e. g. short-running batch queries, long-running stream processing queries, or the acceleration of UDFs (**C2**). These approaches follow different design goals and make different trade-offs between the throughput of the generated code, compilation latency, and ease of use.

Based on our target independent Nautilus IR, Nautilus provides multiple specialized compilation backends, depicted in Table 5.2. These backends receive Nautilus IR fragments of operator pipelines, return executable code, and optimize for specific workloads. Thus, each backend represents a specific spot in the design space and has unique performance characteristics. This flexibility enables the comparison of established query compilation approaches from literature. Currently, Nautilus uses simple heuristics to choose between the backends, e.g., data set size or the presence of UDFs. In the future, we plan to leverage runtime adaptivity in Nautilus to select the optimal execution strategy [162].

Overall Nautilus provides three types of backends: low-latency backends, which minimize compilation time for short-running workloads; high-performance backends, which maximize throughput for long-running workloads; and specialized backends, which concentrate on accelerating specific data processing operations, e.g., the execution of UDFs.

Table 5.1: Comparison of Nautilus's compilation backends.

	Throughput	Latency	Complexity
Low Latency Backend			
Operator Interpreter	▼▼	▲▲	/
Byte Code Interpreter	▼	▲	1200LOC
Flounder	◆	▲	595LOC
MIR	▲	▲	733LOC
High Performance Backend			
MLIR	▲▲	◆	1132LOC
C++	▲▲	▼	495LOC
Specialized Backend			
Babelfish	▲	◆	634LOC

5.5.3.1 Low-Latency Backends

The compilation latency of a query compiler significantly impacts the execution time of short-running queries [162]. To address this issue, Nautilus provides four low-latency backends: a *operator interpreter*, a *byte code interpreter*, *Flounder* [103], and *MIR* [188]. The operator interpreter directly executes Nautilus operators without any tracing, IR generation, or compilation. Thus, it induces no compilation latency ▲▲, but reaches only a very low throughput ▼▼. It is primarily used during development, simplifying the debugging of Nautilus's operators (`C1`). The byte code interpreter, on the other hand, translates the Nautilus IR to a set byte codes as proposed by Kohn et al. [162]. During query execution, it invokes pre-compiled functions for each byte code. As a result, it reaches a higher throughput than the operator interpreter ▼ and only introduces a very low latency for the byte code generation ▲. In contrast, the Flounder [103] backend translates our Nautilus IR directly to machine code. In particular, Flounder is a specialized compiler for data processing workloads and provides a thin abstraction over x64 assembly that performs no additional compiler optimization. This allows Flounder to generate machine code ◆ with negligible compilation times ▲. However, Flounder currently only support x64 platforms and generating machine code for different architectures can require significant engineering effort [121]. Finally, the MIR [188] backend translates our Nautilus IR to the MIR-IR. MIR is a general purpure JIT compiler similar to LLVM, focusing on low compilation times. It was initially developed for the acceleration of Ruby and provides common compiler optimizations, e. g., dead code elimination, instruction combination, and register allocation. In contrast to Flounder, MIR generates more efficient machine code ▲ with similar compilation times ▲.

In summary, our low-latency backends offer trade-offs between compilation time and throughput, enabling Nautilus to support short-running queries efficiently.

5.5.3.2 High-Performance Backends

For long-running queries that process large data sets or streams, it is crucial to maximize execution performance. To this end, Nautilus provides two high-performance code generation backends that aim for optimal code quality: a *MLIR* and a *C++* backend. The MLIR backend translates Nautilus IR to machine code using the MLIR [175] framework. MLIR provides an extensible compiler framework based on LLVM and defines *dialects* to optimize specific

workloads, e.g., parallel loops or reductions. Additionally, LLVM applies optimizations, e.g., auto-vectorization, and enables Nautilus to inline proxy functions in the generated code. As a result, the MLIR backend can generate highly efficient code ▲▲ at a cost of higher compilation latency ♦ (tens of milliseconds). Furthermore, the MLIR backend can integrate 3rd-party dialects, e. g. LingoDB [151] and Daphne [75], to accelerate specific workloads. In contrast, the C++ backend translates Nautilus IR to standard C++ code, which is compiled and linked at runtime. The generated code is easy to debug and comprehensive and also reaches very high throughput ▲▲. Nonetheless, compiling C++ results in considerable latency ▼ [202].

In summary, both high-performance backends produce highly efficient machine code and reach high throughput. As the MLIR backend introduces a lower compilation latency and also provides a higher extensibility, it is the default backend of Nautilus.

5.5.3.3 Specialized Backends

Modern data processing workloads often involve UDFs or ML operations, which cause a high overhead in traditional data processing systems [128]. To this end, Nautilus provides specialized compilation backends that perform specific optimizations to accelerate these workloads. In particular, Nautilus leverages Babelfish, as introduced in Chapter 4, to enable efficient execution of Python, Java, or JavaScript UDFs. Nautilus extends Babelfish and provides support for the Nautilus IR. This enables Babelfish to execute Nautilus operators natively without any data conversion. Furthermore, it allows Babelfish to perform optimization across relational, streaming, and UDF-based operators. For example, it performs inlining to eliminate the boundary between operators. As a result, Babelfish achieves significantly higher throughput for UDF-based workloads ▲▲. However, it also introduces a high compilation latency ▼.

5.5.4 Optimizations

Nautilus's JIT compiler translates operator pipelines to Nautilus IR and provides several compilation backends to target specific workloads. To further improve throughput and code quality, Nautilus can optimizes the generated IR: it reduces the IR complexity, performs constant folding, and inlines runtime code. This is particularly important for low-latency backends as they trade off code quality for a lower compile time. In the following, we provide a brief overview of Nautilus's optimization passes:

Control-Flow Simplification: After tracing, the Nautilus IR resembles the code structure of the operator implementations. Depending on the number of operators, the resulting control flow varies in complexity. In particular, a deep nesting of operators, e.g., due to fused selections or join probes, may result in unnecessary branches and empty basic blocks. In Nautilus's low-latency backends, this introduces additional unnecessary instructions. To mitigate this overhead, Nautilus traverses all basic blocks and checks if their successors are empty. In this case, Nautilus removes the block and adjusts the successors, predecessors, and block arguments.

Instruction Shuffling & Elimination: Based on the simplified IR, Nautilus optimizes the instruction schedule. To this end, it analyzes the instruction within blocks and moves them toward their first usage. This optimization is particularly useful for `load` instructions as they are usually created by a scan operator, even though the loaded value may only be required if a selection predicate qualifies. If instructions have no usage at all, they are removed.

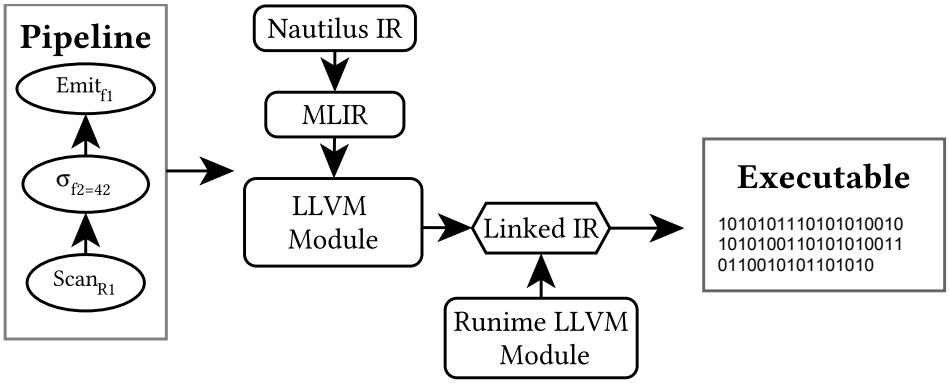


Figure 5.6: Inlining of function calls between generated code and runtime.

Liveness Analysis: In particular, both Nautilus's byte-code interpreter and Flounder backends benefit from liveness information over values in the Nautilus IR. To this end, we traverse the IR and analyze the range between the first and last usage for all instructions, similar to Kohn et al. [162]. Our byte-code interpreter uses this information to reduce the size of its register file, and Flounder reduces the number of stack spills.

Runtime Inlining: As discussed in Section 5.3.2, Nautilus operators, e.g., hash join or aggregations, use function calls to call specific pre-compiled operator logic. Even though these function calls are rather infrequent, they introduce overhead and prevent compiler optimizations. To mitigate this, Nautilus's MLIR backend enables inlining between generated code and pre-compiled runtime code as illustrated in Figure 5.6. For each pipeline, the MLIR backend generates an LLVM IR module, which it links with a pre-compiled set of runtime functions. Based on the combined module LLVM performs optimizations and produces one combined executable.

5.6 Evaluation

In this section, we evaluate Nautilus on a diverse set of workloads. First, we introduce our experimental setup in Section 5.6.1. After that, we conduct two sets of experiments. In Section 5.6.2, we compare the performance of Nautilus across relational, streaming, and UDF-based workloads. In Section 5.6.3, we perform micro-experiments to study specific aspects of Nautilus.

5.6.1 Experimental Setup

Throughout our evaluation, we use the hardware and software configurations that are described in Section 5.6.1.1 and run the workloads that are described in Section 5.6.1.2.

5.6.1.1 Hardware and Software

We execute all experiments on an Intel Xenon Gold 6126 processor with 2.6 GHz and 12 physical cores. Each physical core has a dedicated 32 KB L1 cache for data and instructions. Additionally, each core has 1MB L2 cache, and all cores share a 19.25 MB L3 cache. The test system consists of 755 GB of main memory and runs Ubuntu 22.04. Nautilus's relies on

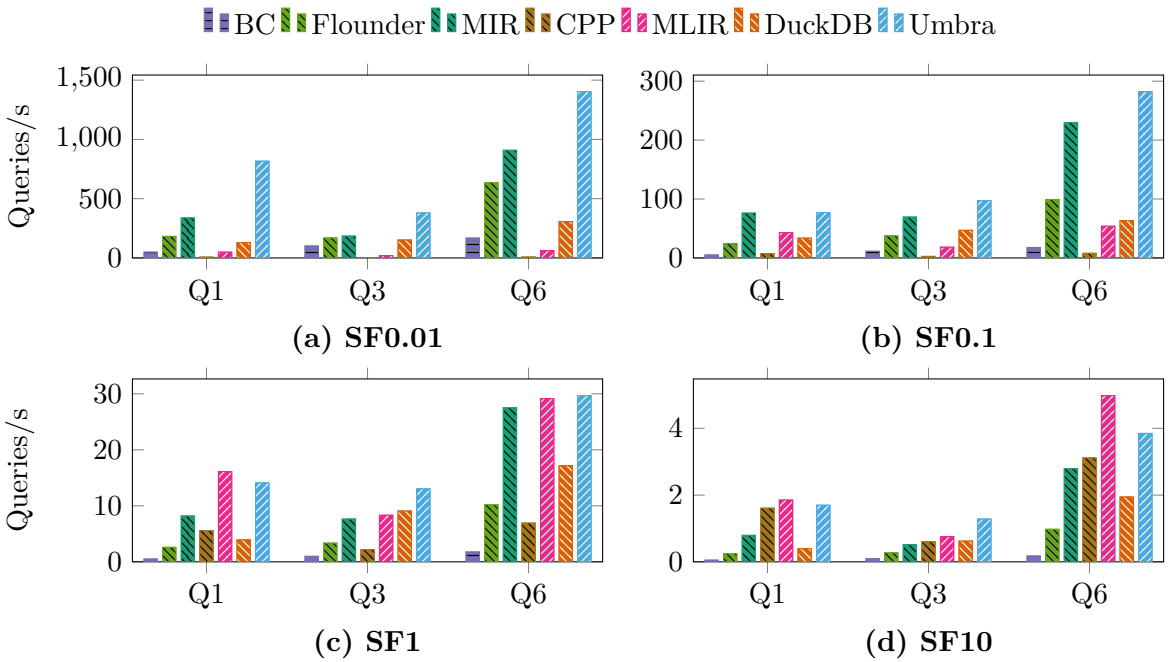


Figure 5.7: Comparison of runtime (compilation time + query execution time) across TPC-H queries 1, 3, and 6 between Nautilus backends, Umbra, and DuckDB.

LLVM-16 and GraalVM 22.3. Furthermore, we use Umbra in version 506343a1a and DuckDB version 0.8.1. If not stated otherwise, we execute all measurements using a single thread.

5.6.1.2 Workloads

Throughout this evaluation, we use the following datasets (stored in memory in a columnar format). To evaluate the OLAP performance, we use queries from the *TPC-H* benchmark with different scale factors. To assess Nautilus's performance on long-running queries, we use the *Yahoo Streaming Benchmark* [67] and the *NexmarkBenchmark* [271] as representative workloads. For UDF-based queries, we use a set of queries, which was used in multiple publications to assess the performance of big data systems on data science workloads [170, 227, 273].

5.6.2 System Comparison

In this set of experiments, we evaluate Nautilus on relational (see Section 5.6.2.1), stream processing (see Section 5.6.2.2), and UDF-based workloads (see Section 5.6.2.3).

5.6.2.1 Relational Workloads

In this experiment, we investigate the performance of Nautilus's compilation backends across TPC-H queries 1, 3 and 6. These queries represent different workload characteristics, e.g., aggregations or joins, and have been used before to assess the efficiency of data processing engines [182]. As baselines, we evaluate all queries on Umbra [203] as a highly optimized query compilation-based system and DuckDB [231] as a representative vectorized system. To assess the impact of compilation latency and code quality on the query execution time across Nautilus's compilation backends, we vary the data size between SF0.01 and SF10.

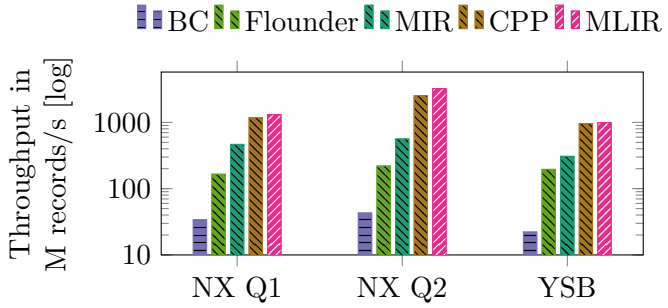


Figure 5.8: Comparison on throughput across stream processing queries between different Nautilus backends.

Results. Figure 5.7 shows the number of executed queries per second for Nautilus's compilation backends in comparison to Umbra and Duckdb. Across all queries and scale factors, Umbra achieves the highest performance as it adaptively switches between a highly optimized low-latency backed and LLVM to generate optimal code [162]. In contrast, DuckDB is outperformed across all queries by up to 6x. For small scale factors (0.01 and 0.1), we observe that Nautilus's Flounder and MIR backends outperform its CPP and MLIR backends as their high compilation latency dominated the query execution time. Furthermore, MIR outperforms Flounder by up to 2x as it produces more efficient code with a similar compilation latency. For scale factors 1 and 10, the impact of compilation latency decreases, and MLIR outperforms MIR and Flounder and reaches the performance of Umbra. On Query 6 and scale factor 10, the MLIR backend produces SIMD code and outperforms Umbra by 1.2x. In contrast, Nautilus is not competitive on Query 3 as it currently only provides a naive chained hash-table implementation that causes cache misses.

Summary. This experiment shows that depending on the scale-factor Nautilus's, compilation backends can reach comparable performance to Umbra's highly optimized query compiler. However, no backend is optimal across all scale factors. Thus, an adaptive approach as proposed by Kohn [162] is required to support different workloads efficiently.

5.6.2.2 Stream Processing Workloads

In this experiment, we evaluate Nautilus's compilation backends across stream processing workloads. As these are long-running queries, the compilation latency becomes neglectable. Thus we only assess execution performance in the number of records processed per second. We evaluate the following three queries: NX1 and NX2 from the Nexmark benchmark perform selection and map operations. In contrast, the YSB query performs a selection and a keyed aggregation over a tumbling time-based window of 10 seconds.

Results. Figure 5.7 shows the throughput of the executed queries across Nautilus's compilation backends. Across all queries, we observe that Nautilus's high-performance backends, MLIR and CPP, achieve the highest performance. Both backends provide sophisticated compiler optimizations, e.g., auto-vectorization and loop-unrolling, to produce efficient code. Interestingly, the MLIR outperforms CPP by up to 1.2x even though both use LLVM as a compiler. This indicates that MLIR, as an intermediate representation, enables additional compiler optimizations compared to CPP code. Furthermore, our results show that Nautilus's low-latency backends are outperformed significantly on long-running workloads.

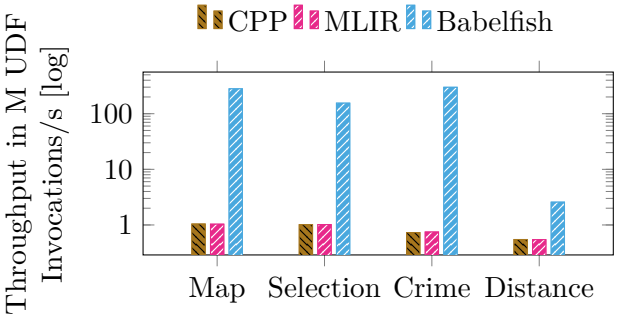


Figure 5.9: Comparison on throughput on UDF-based workloads across Nautilus backends and the Babelfish accelerator.

Summary. This experiment showed that the additional compiler optimizations of Nautilus's high-performance backends are crucial to reach peak performance for long-running workloads. For this class of workloads, compilation latency is neglectable, and to maximize high code quality is desirable.

5.6.2.3 UDF-based Workloads

In this experiment, we evaluate the impact of Nautilus's special purpose Babelfish backend across four queries that combine relation operators with one or more Java UDFs with different workloads characteristics. The first two queries, i.e., *map* and *selection*, involve computationally simple UDFs and assess the UDF invocation overhead between the data processing system and the UDF runtime. The third query calculates the average crime index for a set of cities and involves multiple UDFs. In contrast, the last query calculates the distance between two points using Vincenty's formula [276] and is computationally intensive.

Results. Figure 5.9 shows the throughput of Nautilus's high-performance backends, MLIR and CPP, in comparison to its Babelfish backend across all four UDF-based queries. Overall, we can make two key observations. First, the CPP and the MLIR backends achieve the same low performance as the UDF overhead dominates the execution time of both. As discussed in Chapter 4.3.1, this involves the invocation of the UDF runtime, the data exchange between the host system and the UDF, and the data conversion. Second, Babelfish eliminates this overhead and achieves a speedup of up to two orders of magnitude. In contrast to the Nautilus's high-performance backends, Babelfish executes relational operators and UDFs in the same engine, which enables holistic optimizations across operator boundaries. This approach is even beneficial for computational intensive UDF, i.e., the distance query, as it enables further optimization by Babelfish's JIT compiler.

Summary. This experiment showed that specialized compilation backends like Babelfish have a significant performance benefit for specific workloads. Babelfish processes up to two orders of magnitudes more tuples per second than Nautilus's high-performance backends and enables efficient UDF processing.

5.6.3 Micro Experiments

In this section, we conduct a set of micro experiments to assess specific aspects of Nautilus. First, we analyze the compilation latency of Nautilus's backends for different queries in Section

Table 5.2: Comparison of compilation latency in milliseconds across Nautilus's and Umbra compilation backends for TPC-H Queries 1, 3, and 6.

	Nautilus					Umbra	
	BC	**MIR**	**Flounder**	**MLIR**	**CPP**	**Low-Latency**	**LLVM**
Q1							
Tracing	0.49	0.51	0.50	0.56	1.79	-	-
IR Generation	0.13	0.12	0.16	0.16	0.48	-	-
Lowering	0.06	0.24	0.17	1.54	0.70	-	-
Code Generation	-	0.85	0.37	24	131	-	-
Σ Compilation	0.69	1.72	1.18	27.35	134	0.62	32.74
Q3							
Tracing	1.05	1.08	1.07	1.24	4.20	-	-
IR Generation	0.30	0.27	0.26	0.29	1.07	-	-
Lowering	0.16	0.5	0.04	3.99	1.64	-	-
Code Generation	-	2.1	0.76	59.66	377	-	-
7 Σ Compilation	1.51	3.98	2.52	65.16	381	0.80	40.53
Q6							
Tracing	0.19	0.20	0.19	0.23	0.76	-	-
IR Generation	0.04	0.04	0.04	0.04	0.17	-	-
Lowering	0.03	0.09	0.06	1.13	0.27	-	-
Code Generation	-	0.38	0.11	0.19	122	-	-
Σ Compilation	0.27	0.74	0.54	20.59	123	0.32	13.98

5.6.3.1. Second, we study the impact of the query complexity on the compilation latency to investigate the robustness of individual backends in Section 5.6.3.2. Finally, we investigate the impact of runtime inlining in Section 5.6.3.3.

5.6.3.1 Compilation Latency

In this experiment, we examine the compilation latency of Nautilus's compilation backends for the TPCH Queries 1, 3, and 6. To this end, we report the cumulated latency and analyze the latency of individual compilation phases: *1)* Initial tracing. *2)* Generation of Nautilus-IR. *3)* Lowering to the IR of a specific compilation backend. *4)* Final code generation. As a reference, we also report the compilation latency of Umbra's low-latency and LLVM-based compilation backends for all queries.

Results. Table 5.2 shows the latency breakdown of Nautilus's compilation backends in comparison to Umbra across the selected TPCH queries. For Nautilus we observe that its low-latency backends compile queries in 0.27 to 3.9ms. Among these, Nautilus's bytecode interpreter (BC) reaches the lowest latency as it avoids any code generation. In contrast, Flounder and MIR translate the Nautilus IR to machine code. As MIR performs more optimizations, it requires up to 3x more time to generate code in comparison to Flounder. On the other side of the spectrum, Nautilus's high-performance backends induce a 35x to 100x higher compilation latency. Furthermore, our analysis reveals that Nautilus's tracing approach requires 0.2ms to 1.24ms per query. As a result, it has a high influence on the cumulated compilation time of low-latency backends (up to 60%) but is negligible for high-performance backends. For these, the final code generation dominates the overall compilation time. In this case, the CPP backend requires up to 10x more time than the MLIR backend. In comparison

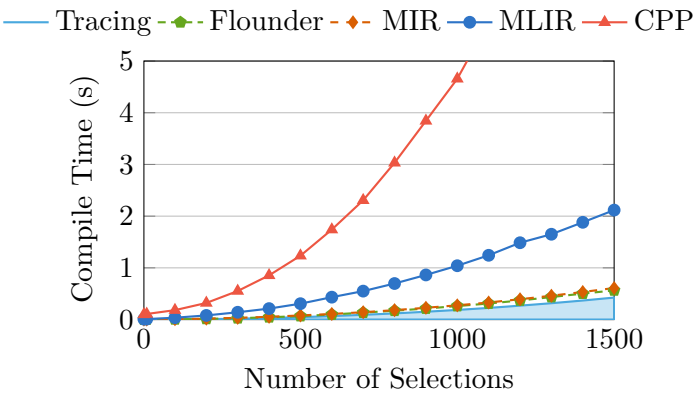


Figure 5.10: Compilation time for Tracing, Flounder, MIR, MLIR, and CPP in comparison to the code complexity (number of selections within one operator pipeline).

to Umbra, we can make the following two observations. First, Umbra's low-latency compilation backend achieves compilation times that are, on average, 1.4 times faster than those of Nautilus. These low compilation times are a significant factor for Umbra's superior performance on small scale-factors in the experiments of Section 5.6.2.1. Second, Umbras LLVM-based backend reaches similar compilation times as Nautilus's MLIR backend. The variations between these two backends can primarily be attributed to different compiler versions and settings.

Summary. In this experiment, we investigated the compilation latency of Nautilus's compilation backends. We saw that Nautilus's low-latency backends reach significantly lower compilation times than its high-performance backends. Furthermore, our results indicate that Nautilus is able to reach similar compilations times as Umbra. However, currently tracing induces an overhead for very short-running queries, which have a sub-millisecond execution time. For these workloads, a further reduction of compilation latency is beneficial.

5.6.3.2 Compilation Latency Robustness

In this experiment, we analyze the impact of the query complexity on the latency of Nautilus's compilation backends. To this end, we assess the compilation latency for queries with 1 to 1500 selections as additional selections increase the control-flow nesting of the Nautilus-IR.

Results. Figure 5.9 shows the latency of all compilation backends for the increasing number of selections. As all backends require the initial tracing phase, we indicate the trace latency in blue. The time for tracing increases linearly with the number of selections, as discussed in Section 5.5.1, and reaches 400ms for 1500 selections. In comparison, Nautilus's low-latency backends only introduce a short additional compilation time. Even for a high number of selections, the latency of Flunder and MIR stays similarly low. In contrast, CPP and MLIR require significantly more time and introduce an overhead of multiple seconds.

Summary. In this experiment, we showed that Nautilus's compilation backends scale even to large queries. However, its high-performance backends introduce a significant compilation latency. To handle such queries, Nautilus could either rely on its low-latency backends or split complex queries in multiple pipelines, which are easier to compile.

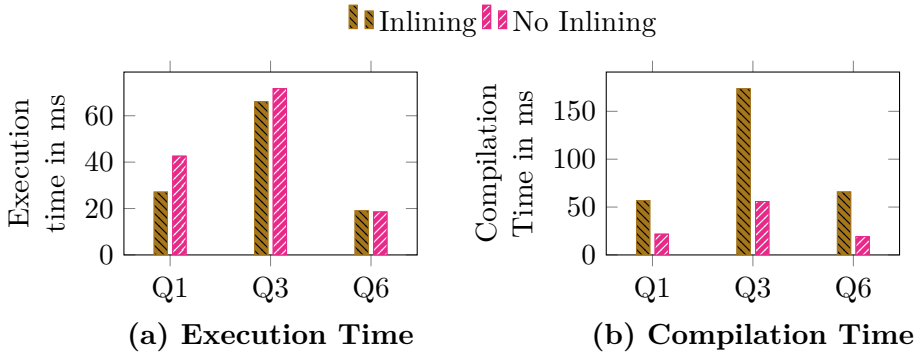


Figure 5.11: Impact of runtime inlining for Nautilus's MLIR backend.

5.6.3.3 Impact of Runtime Inlining

In this micro experiment, we assess the impact of runtime inlining for Nautilus's MLIR backend as proposed in Section 5.5.4. This optimization eliminates function calls from the generated code, for example, to data structures, and enables further compiler optimizations. To assess the impact of this optimization, we report the execution and compilation time of Nautilus for TPCH Q1, Q3, and Q6 with and without activated inlining.

Results. Figure 5.11 shows the impact of inlining on the execution time (a) and compilation time (b). For queries 1 and 3 inlining reduces the execution time by up to 20% as both queries involve function calls to hash-tables that can be inlined. After inlining, the compilation backend can perform further optimizations like loop unrolling, which are not performed in the presents of function calls. In contrast, inlining does not impact the execution time of Query 6 as it contains no function calls. In addition to the execution time, inlining significantly impacts the compilation time, which increases by up to 2.4x.

Summary. In this experiment, we showed that runtime inlining can significantly impact the execution time of queries. However, it also has a high compilation time overhead. As a result, it should only be enabled for long-running queries.

5.6.4 Discussion

Our evaluation has shown that Nautilus provides efficient code generation for various data processing workloads. Its compilation backends provide low compilation latency for short-running queries, achieve high performance for long-running queries, and accelerate special workloads like UDFs. Furthermore, Nautilus reaches comparable performance to highly-optimized compilation-based engines like Umbra while it provides a high-level operator implementation interface. This enables engineers to focus on developing features and relieves them of the complexity of traditional query compilation approaches. Finally, we demonstrated that Nautilus even reaches high performance on complex workloads and can provide further performance improvements through inlining for very long-running queries.

5.7 Related Work

Over the last decade, query compilation was extensively studied [166, 202, 234] and implemented across various data processing systems [6, 102, 166, 195, 202, 203, 215, 217, 278]. Generally, Nautilus can be distinguished from prior works in three directions, i.e., work on *code generation abstractions*, work on *low-latency query compilation*, and work that applies query compilation to *diverse workloads*. In the following, we discuss these directions individually.

Code generation abstractions. The first line of research proposed interfaces and abstractions to reduce the complexity of compilation-based query execution engines [10, 18, 45]. Many query compilers generate code in programming languages like C++, Java, or OpenCL to make the generated code easy to debug [45]. However, this often leads to significant compilation latencies. For example, Amazon Redshift employed a global cache to mitigate compilation overhead [18]. Nautilus, however, decouples the implementation of operators and their execution. Engineers can debug operators directly while the compiler translates them to efficient code at runtime. Similarly, researchers proposed Domain Specific Languages (DSLs), like LMS [250], Voila [129], Weld [214, 215], or Voodoo [225], to decouple implementation and execution. These DSLs introduce declarative primitives that represent specific data processing operations, e.g., `hash`, `bucket_insert` in Voila [129]. However, these do not map directly to imperative implementations, which hinders testing and debugging. Additional work, proposed programming interfaces for code generation [133, 157]. Although this hides the details of code generation, it does not address the fundamental mismatch between the implementation of operators that generate code and the executed code at runtime. In contrast, Nautilus's operators and data structures correspond directly to C++ code, simplifying development, testing, and maintenance. Recent work proposed using MLIR [175] as a framework for query compilation in data processing systems [75, 151]. These approaches introduce specialized MLIR dialects to model primitive data processing operations similar to Voila [129]. In contrast, Nautilus uses MLIR as a compilation backend for Nautilus IR. To this end, it only relies on standard dialects, which simplifies our MLIR integration.

Low-latency Query Compilation. The second research area focused on techniques to reduce the latency of query compilers [103, 133, 157, 162]. To this end, Kohn et al. [162] proposed bytecode interpretation, Funke et al. [103] and Kersten et al. [157] proposed special purpose compilers, and Haffner et al. [133] proposed to target JIT-compiler like V8. Nautilus incorporates these approaches and provides different compilation backends using bytecode interpretation, special-purpose compilers, and low-latency JIT compilers. This enables Nautilus to compare the individual techniques and to target a wide range of workloads, including short-running queries. To this end, Nautilus selects a specific compilation backend depending on the workload and hardware characteristics. This allows Nautilus to support x64 as well as ARM CPU architectures and relieves engineers from the complexity of developing query compilers from scratch. In particular, recent work showed that porting and maintaining custom low-latency compilers across different architectures can be very challenging [121].

Diverse Workloads. The third line of research leverages query compilation to accelerate different data processing workloads, e.g., stream processing [31, 125, 147, 265, 289], spatial data processing [260], machine learning [75], and polyglot queries involving UDFs [71, 128, 251, 253].

Nautilus integrates many aspects of these works to target a wide range of workloads. For example, it adapts the efficient operators for stream processing from Grizzly (see Chapter 3) and provides additional compiler optimizations to increase execution performance, e. g., runtime inlining. Furthermore, Nautilus leverages Babelfish (see Chapter 4) as a special-purpose compilation backend for UDFs. To this end, it extends Babelfish and enables holistic optimization across Nautilus operators and UDFs. The support of these workloads underpins the flexibility of Nautilus's operator interface and its compilation approach.

5.8 Conclusion

In this chapter, we have presented Nautilus, a framework to bridge the gap between query interpretation and compilation. To this end, Nautilus addresses two crucial challenges of current query compilation approaches. First, operators in compilation-based engines are hard to implement as they generate code at runtime. To this end, Nautilus provides an interface that enables system engineers to implement operators in imperative code that is easy to develop, debug, and maintain. Second, research proposed a variety of query compilation approaches that optimize for specific workloads. In contrast, Nautilus proposes a trace-based JIT compiler to decouple the implementation of operators from their execution. At execution time, it provides multiple compilation backends with different performance characteristics to efficiently support specific data processing workloads. As a result, Nautilus compiles queries to efficient code without sacrificing the productivity of engineers. Thus, Nautilus enables engineers to focus on the implementation of features instead of handling the complexity of query compilation. Furthermore, our evaluation show that Nautilus can achieve high performance across various workloads and reaches the performance of optimized state-of-the-art query compilers.

In general, Nautilus makes query compilation more accessible to a broader audience. To this end, we provide Nautilus as an open framework that is easy to integrate with different data processing systems. This reduces the engineering effort needed to develop compilation-based execution engines and helps researchers to focus on data-processing related challenges. To demonstrate the practicality of Nautilus, we integrated it in the execution engine of NebulaStream [289].

5.8.1 Integration in NebulaStream

NebulaStream targets a variety of workloads and provides operators for batch-, stream-, and complex-event-processing. Using Nautilus, we re-implemented its query execution engine and migrated its operators. In this process, we extended Nautilus in the following dimensions:

Parallelization-aware Operators. As we have discussed in Chapter 3, NebulaStream leverages task-based parallelization to fully utilize multi-core CPUs. In this execution strategy, stateful operators, e.g., aggregation and join, are parallelization-aware and operate on a shared state. Thus, concurrent state accesses require synchronization. To this end, we introduced common patterns and data structures that can be used across different stateful operators. For example, NebulaStreams batch and stream joins rely on the same underlying data structures.

Portability. As NebulaStream targets a variety of different execution environments, we improved the portability of Nautilus. In particular, we improved support for Linux and

MacOS as well as x64 and ARM architectures. Depending on the specific properties of a NebulaStream Worker, Nautilus selects the most suited execution strategy. Furthermore, Nautilus's compilation backends leverage this information to perform hardware-specific optimizations. For instance, the MLIR-backed leverages AVX SIMD instructions if available.

In summary, Nautilus allows NebulaStream to benefit from the efficiency of query compilation without compromising productivity. The migration to Nautilus demonstrated the ease of use of the proposed framework and enabled NebulaStream to target also short-running workloads. This also improves the testability of the systems, as short-running unit tests execute significantly faster. Furthermore, Nautilus reduced the implementation complexity of NebulaStream operators. Now even undergrad students can make valuable contributions.

6

Additional Contributions

In this chapter, we describe further research contributions which have been made while working on this thesis. The following additional contributions are not part of the thesis contents but are closely related to the topics presented in this thesis:

NebulaStream: Through the thesis, we used NebulaStream as the target system for our contributions. NebulaStream is a general-purpose data management platform that focuses on highly distributed IoT environments. Thus the scope and vision of the system exceeds the topic of this thesis. However, the author made several key contributions to the execution engine of the system. In particular, the results of this thesis lay the foundation for NebulaStream's architecture and enable the efficient execution of various data processing workloads. Furthermore, the author was involved in the following publications that discuss individual aspects of NebulaStream. In *The NebulaStream Platform: Data and Application Management for the Internet of Things* [289], *NebulaStream: Complex Analytics Beyond the Cloud* [293], and *NebulaStream: Data Management for the Internet of Things* [288] we presented the overall project vision, NebulaStream's system architecture, and its target environment. In *ExDRa: Exploratory Data Science on Federated Raw Data* [27] we leveraged NebulaStream for data gathering in real-world data science pipeline in combination with SystemDS [37] and in *Showcasing Data Management Challenges for Future IoT Applications with NebulaStream* [181] we present an interactive smart city simulation framework to illustrate research challenges of NebulaStream. Finally, we extend in *Architecting a Dynamic and Efficient Stream Processing Engine* [247] our work on Grizzly (see Chapter 3) and discuss how NebulaStream schedules concurrent and dynamic workloads.

1. **Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M. Grulich, Sebastian Breß, Jonas Traub, Volker Markl**: *The NebulaStream Platform: Data and Application Management for the Internet of Things.* CIDR, 2020.

2. **Steffen Zeuch, Eleni Tzirita Zacharatou, Shuhao Zhang, Xenofon Chatziliadis, Ankit Chaudhary, Bonaventura Del Monte, Dimitrios Giouroukis, Philipp M. Grulich, Ariane Ziehn, Volker Markl**: *NebulaStream: Complex Analytics Beyond the Cloud.* Open Journal of Internet Of Things, 2020.

3. **Sebastian Baunsgaard, Matthias Boehm, Ankit Chaudhary, Behrouz Derakhshan, Stefan Geißelsöder, Philipp M. Grulich, Michael Hildebrand, Kevin Innerebner, Volker Markl, Claus Neubauer, Sarah Osterburg, Olga Ovcharenko, Sergey Redyuk, Tobias Rieger, Alireza Rezaei Mahdiraji, Sebastian Benjamin Wrede, Steffen Zeuch**: *ExDRa: Exploratory Data Science on Federated Raw Data.* ACM SIGMOD, 2021.

4. **Steffen Zeuch, Xenofon Chatziliadis, Ankit Chaudhary, Dimitrios Giouroukis, Philipp M. Grulich, Dwi Nugroho, Ariane Ziehn, Volker Markl**: *NebulaStream: Data Management for the Internet of Things.* Datenbank-Spektrum, 2022.

5. **Aljoscha Lepping, Hoang Mi Pham, Laura Mons, Balint Rueb, Ankit Chaudhary, Philipp M. Grulich, Steffen Zeuch, Volker Markl**: *Showcasing Data Management Challenges for Future IoT Applications with NebulaStream.* PVLDB, 2023.

6. **Nils L. Schubert, Philipp M. Grulich, Bonaventura Del Monte, Steffen Zeuch, Volker Markl**: *Architecting a dynamic and efficient stream processing engine.* Under Submission.

Stream Processing Techniques: In addition to the topics of this thesis, we made several contributions to the field of stream processing. In *Scotty: General and Efficient Open-source Window Aggregation for Stream Processing Systems* [269] and *Algorithms for Windowed Aggregations and Joins on Distributed Stream Processing Systems* [274] we proposed efficient approaches for the implementation of window aggregation using stream slicing. In *Disco: Efficient Distributed Window Aggregation* [29], we extended this approach towards aggregation in distributed topologies. Both works are now integrated into NebulaStream. In addition, we provided in *Generating Reproducible Out-of-Order Data Streams* [126] a data generator for syntactic out-of-order streams to simulate a variability of stream characteristics. Furthermore, we provided in a *Survey of window types for aggregation in stream processing systems* [275] a comprehensive overview of window types, their formal definitions, and use-case examples. Finally, we investigated in *Bridging the Gap: Complex Event Processing on Stream Processing Engines* [301] techniques to unify both traditional analytical stream processing operators and pattern detection operators from complex event processing.

1. **Philipp M. Grulich, Jonas Traub, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, Volker Markl**: *Generating Reproducible Out-of-Order Data Streams.* ACM DEBS, 2019.

2. **Lawrence Benson, Philipp M. Grulich, Steffen Zeuch, Volker Markl, Tilmann Rabl**: *Disco: Efficient Distributed Window Aggregation.* EDBT, 2020.

3. **Jonas Traub, Philipp M. Grulich, Alejandro Rodriguez Cuellar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, Volker Markl**: *Scotty: General and Efficient Open-source Window Aggregation for Stream Processing Systems.* ACM TODS, 2021.

4. **Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, Volker Markl**: *Algorithms for Windowed Aggregations and Joins on Distributed Stream Processing Systems.* Datenbank-Spektrum, 2022.

5. **Juliane Verwiebe, Philipp M. Grulich, Jonas Traub, Volker Markl**: *Survey of window types for aggregation in stream processing systems.* VLDB Journal, 2023.

6. **Ariane Ziehn, Philipp M. Grulich, Steffen Zeuch, Volker Markl**: *Bridging the Gap: Complex Event Processing on Stream Processing Engines.* Under Submission.

Modern Hardware: Finally, we investigated several aspects of data processing on modern hardware. In *Parallelizing Intra-Window Join on Multicores: An Experimental Study* [296], we provided a comprehensive analysis of different approaches for the parallelization of window joins in stream processing systems on modern multicore CPUs. Based on this, we extended this analysis in *Benchmarking Stream Join Algorithms on GPUs: A Framework and its Application to the State-of-the-art* [206] and investigated the effectiveness of GPUs for the acceleration of window joins. In *An Energy-Efficient Stream Join for the Internet of Things* [197], we proposed a novel window stream join to reduce the energy consumption of embedded system-on-a-chip platforms that provide integrated GPUs. Finally, we present in *Exploiting Access Pattern Characteristics for Join Reordering* [246] a novel approach for the adaptive reordering of joins in batch systems using non-invasive hardware performance counters.

1. **Shuhao Zhang, Yancan Mao, Jiong He, Philipp M. Grulich, Steffen Zeuch, Bingsheng He, Richard T. B. Ma, Volker Markl**: *Parallelizing Intra-Window Join on Multicores: An Experimental Study.* ACM SIGMOD, 2021.

2. **Adrian Michalke, Philipp M. Grulich, Clemens Lutz, Steffen Zeuch, Volker Markl**: *An Energy-Efficient Stream Join for the Internet of Things.* DaMoN, 2021.

3. **Nils L. Schubert, Philipp M. Grulich, Steffen Zeuch, Volker Markl**: *Exploiting Access Pattern Characteristics for Join Reordering.* DaMoN, 2023.

4. **Dwi Nugroho, Philipp M. Grulich, Steffen Zeuch, Volker Markl**: *Benchmarking Stream Join Algorithms on GPUs: A Framework and its Application to the State-of-the-art.* EDBT, 2024.

7

Conclusion and Future Research

Throughout this thesis, we revisited the architecture of query execution engines to provide efficient data processing in diverse environments. Such environments consist of complex analytical pipelines that exceed traditional OLAP workloads and combine relational operators with stream processing and UDF-based operators. To this end, we adopted query compilation and proposed building blocks that lay the foundation for a new generation of data processing engines that provide high efficiency, flexibility, and robustness, across various workloads.

In particular, we made the following contributions. In Chapter 3, we first focused on stream processing workloads and introduced adaptive query compilation. Our approach addresses the unique semantics of stream processing and translates operators into efficient code. At runtime, it monitors data characteristics of long-running queries, detects changes, and deploys optimized code variants. This improves the efficiency of long-running stream processing workloads significantly. In Chapter 4, we leveraged query compilation to optimize polyglot queries that combine relational operators and UDFs in high-level programming languages like Python, Java, or JavaScript. In particular, we introduced holistic optimizations that eliminate the boundary between polyglot operators to mitigate the overhead of UDFs. Based on both approaches, we proposed in Chapter 5 a novel framework for the development of compilation-based execution engines. This framework reduces the development complexity of query compilers without sacrificing performance. In particular, our framework provides different query execution strategies to reach high performance for both short-running and long-running workloads. Furthermore, it incorporates our earlier work and provides efficient support for batch, stream, and UDF-based operators. Finally, we integrated our contributions into NebulaStream to demonstrate the practicality of our solutions. In particular, our work enables NebulaStream to fully utilize modern hardware and reach high performance.

Overall, this thesis expands the applicability of query compilation and demonstrates its benefits across diverse workloads and use cases. Our work lays the foundation for a hardware-conscious architecture of future data management systems and provides building blocks that enable significant efficiency improvements while also ensuring maintainability. This is especially important for managed data processing platforms in the cloud that must reach high performance across diverse workloads and provide high productivity for system engineers.

Using our contributions, cloud vendors can achieve these requirements and specialize query execution towards specific workload characteristics. This reduces costs and increases profit. As a result, our work unlocks future data-intensive applications that process growing data volumes at increasing velocity.

Future Research

This thesis lays the foundation for the broader adoption of query compilation in future data processing systems that target diverse environments. Our contribution enables the efficient execution of polyglot queries that involve relational, stream processing, and UDF-based operators in the same engine. This reduces development costs and increases the system's usability. Furthermore, our work opens additional research challenges for future work:

Distributed Stream Processing. In this thesis, we demonstrated that query compilation significantly improves the performance of scale-up SPSs. In particular, our approach increases hardware utilization and enables the system to reach high throughput. However, in many scenarios, the network bandwidth of individual nodes limits the ingestion rate. To this end, recent work suggested the combination of scale-up and scale-out architectures [31]. Such a system distributes work across nodes to mitigate bandwidth bottlenecks and could integrate our query compilation approach to optimize execution on individual nodes. This line of work proposed, efficient window aggregation approaches [29, 284] that can distribute stream processing workloads in hierarchical network topologies. Further network bottleneck could be mitigated by integrating our compilation-based engine with recent work on the exploitation of high-performance networks using RDMA [78, 300].

Adaptive Optimizations. Our work demonstrated that adaptive optimizations are crucial to process diverse workloads efficiently. Exploiting these optimizations for stream processing opens a potential for additional research. For example, applying statistical models, such as Adwin [33, 124], which detect shifting data characteristics, could potentially generalize our method for a broader range of optimizations. Additionally, we could integrate work on self optimizing data structures [59, 143] to further specialize stateful stream processing operators to specific data and hardware characteristics. Moreover, our work could be combined with recent work in adaptive optimizations within compilation-based batch processing systems [130, 195] to reduce re-optimization overhead. Finally, there is the need for dedicated cost models to select stream processing-specific query optimization [138]. To this end, recent work proposed learned cost-models to predict specific parameters of stream processing systems [7, 8].

Polyglot Query Processing. In this work, we showed that holistic optimizations enable query-compilation-based engines to improve the efficiency of polyglot queries significantly. Our work could be used as a blueprint to accelerate the execution of polyglot operators across various data processing systems. In particular, it would be interesting to provide UDF execution as a 3-party extension for existing systems. In this case, mitigating communication overhead between the host system and the UDF runtime remains challenging [127].

Machine Learning Workloads. In this thesis, we demonstrated that our compilation-based engine can optimize queries across workload boundaries. One increasingly important workload categories are linear algebra and tensor operations [37, 38]. In particular, it would

be interesting to investigate holistic optimizations that take these operations into account and improve the efficiency of training as well as inference pipelines. Examples include the operation on compressed data [26], the integration of compilers that focus on ML operations [75, 159], or the combination of relation and tensor operators [169].

Heterogeneous Hardware. Throughout this thesis, we focused on improving the efficiency of query execution on modern hardware. An important aspect in this direction is the exploitation of specialized hardware accelerators, e.g., wide SIMD instructions, GPUs, or TPUs. To this end, we could extend the Nautilus query compiler. In particular, we could investigate the portability of operator implementation across accelerators [45, 225], the scheduling of tasks and data between devices [83], and to leverage hardware-aware operator implementations [185]. One example could be the extension of Nautilus MLIR backend, which already offer abstractions to target hardware heterogeneity [175]. Additionally, modern-hardware could be leveraged to accelerate complex stream processing operators. For example, Darwin [31] integrated Viper [30] leverage persistent-memory as a state-backed for stateful operators.

In summary, we envision that future work will leverage query-compilation to provide high performance for a wide range of workloads and optimize code of specific hardware.

List of Figures

List of Tables

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. **The design of the borealis stream processing engine**. In *CIDR*, 5, pages 277–289, 2005. https://www.cidrdb.org/cidr2005/papers/P23.pdf.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. **Aurora: a new model and architecture for data stream management**. *VLDB Journal*, 12(2):120–139, 2003. doi:10.1007/s00778-003-0095-z.

[3] S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky. **Jet: An embedded DSL for high performance big data processing**. In *BigData*, 2012.

[4] D. Adams. *The Hitchhiker's Guide to the Galaxy*. Pan Books, 1979.

[5] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. **Firecracker: Lightweight Virtualization for Serverless Applications**. In *NSDI*, 20, pages 419–434, 2020. https://www.usenix.org/conference/nsdi20/presentation/agache.

[6] S. Agarwal, D. Liu, and R. 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. [Online; accessed 31.5.2019].

[7] P. Agnihotri, B. Koldehofe, C. Binnig, and M. Luthra. **PANDA: Performance Prediction for Parallel and Dynamic Stream Processing**. In *Proceedings of the 16th ACM International Conference on Distributed and Event-Based Systems*, DEBS '22, page 180–181, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3524860.3543281, doi:10.1145/3524860.3543281.

[8] P. Agnihotri, B. Koldehofe, C. Binnig, and M. Luthra. **Zero-Shot Cost Models for Parallel Stream Processing**. In *Proceedings of the Sixth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '23, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3593078.3593934.

[9] D. Agrawal, S. Chawla, B. Contreras-Rojas, A. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, S. Thirumuruganathan, and A. Troudi. **RHEEM: enabling cross-platform data processing: may the big data be with you!** *PVLDB*, 11(11):1414–1427, 2018. doi:10.14778/3236187.3236195.

[10] Y. Ahmad and C. Koch. **DBToaster: A SQL Compiler for High-Performance Delta Processing in Main-Memory Databases**. *PVLDB*, 2(2):1566–1569, aug 2009. doi:10.14778/1687553.1687592.

[11] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. **DBMSs on a modern processor: Where does time go?** In *VLDB*, pages 266–277, 1999.

[12] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. **MillWheel: Fault-Tolerant Stream Processing at Internet Scale**. *PVLDB*, 6(11):1033–1044, 2013. doi:10.14778/2536222.2536229.

[13] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. **The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing**. *PVLDB*, 8(12):1792–1803, 2015. doi:10.14778/2824032.2824076.

[14] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl. **Implicit parallelism through deep language embedding**. In *SIGMOD*. ACM, 2015. doi:10.1145/2949741.2949754.

[15] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. **Conversion of control dependence to data dependence**. In *SIGPLAN*, pages 177–189. ACM, 1983.

[16] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia. **Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark**. In *SIGMOD*. ACM, 2018. doi:10.1145/3183713.3190664.

[17] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. **Spark SQL: Relational Data Processing in Spark**. In *SIGMOD*, pages 1383–1394. ACM, 2015. doi:10.1145/2723372.2742797.

[18] N. Armenatzoglou, S. Basu, N. Bhanoori, M. Cai, N. Chainani, K. Chinta, V. Govindaraju, T. J. Green, M. Gupta, S. Hillig, E. Hotinger, Y. Leshinksy, J. Liang, M. McCreedy, F. Nagel, I. Pandis, P. Parchas, R. Pathak, O. Polychroniou, F. Rahman, G. Saxena, G. Soundararajan, S. Subramanian, and D. Terry. **Amazon Redshift Re-Invented**. In *SIGMOD*, page 2205–2217. ACM, 2022. doi:10.1145/3514221.3526045.

[19] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. Gray, P. P. Griffiths, W. F. K. III, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. **System R: Relational approach to database management**. *TODS*, 1(2):97–137, 1976. doi:10.1145/320455.320457.

[20] D. I. August, W.-m. W. Hwu, and S. A. Mahlke. **A framework for balancing control flow and predication**. In *MICRO*, pages 92–103. IEEE, 1997.

[21] S. Babu and P. Bizarro. **Adaptive query processing in the looking glass**. In *CIDR*, 2005. http://cidrdb.org/cidr2005/papers/P20.pdf.

[22] S. Babu, P. Bizarro, and D. DeWitt. **Proactive re-optimization**. In *SIGMOD*, pages 107–118. ACM, 2005. doi:10.1145/1066157.1066171.

[23] V. Bala, E. Duesterwald, and S. Banerjia. **Dynamo: A Transparent Dynamic Optimization System**. In *PLDI*, page 1–12. ACM, 2000. doi:10.1145/349299.349303.

[24] T. Bang, N. May, I. Petrov, and C. Binnig. **The tale of 1000 cores: An evaluation of concurrency control on real (ly) large multi-socket hardware**. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, pages 1–9, 2020.

[25] S. Basinasetty. **User Defined Functions in SQL**, 2021. https://www.tutorialgateway.org/user-defined-functions-in-sql/.

[26] S. Baunsgaard and M. Boehm. **AWARE: Workload-Aware, Redundancy-Exploiting Linear Algebra**. *Proc. ACM Manag. Data*, 1(1), may 2023. https://doi.org/10.1145/3588682, doi:10.1145/3588682.

[27] S. Baunsgaard, M. Boehm, A. Chaudhary, B. Derakhshan, S. Geisselsöder, P. M. Grulich, M. Hildebrand, K. Innerebner, V. Markl, C. Neubauer, S. Osterburg, O. Ovcharenko, S. Redyuk, T. Rieger, A. R. Mahdiraji, S. B. Wrede, and S. Zeuch. **ExDRa: Exploratory Data Science on Federated Raw Data**. In *SIGMOD*, pages 2450–2463. ACM, 2021. doi:10.1145/3448016.3457549.

[28] A. Behm, S. Palkar, U. Agarwal, T. Armstrong, D. Cashman, A. Dave, T. Greenstein, S. Hovsepian, R. Johnson, A. S. Krishnan, P. Leventis, A. Luszczak, P. Menon, M. Mokhtar, G. Pang, S. Paranjpye, G. Rahn, B. Samwel, T. van Bussel, H. V. Hovell, M. Xue, R. Xin, and M. Zaharia. **Photon: A Fast Query Engine for Lakehouse Systems**. In *SIGMOD*, pages 2326–2339. ACM, 2022. doi:10.1145/3514221.3526054.

[29] L. Benson, P. M. Grulich, S. Zeuch, V. Markl, and T. Rabl. **Disco: Efficient Distributed Window Aggregation**. In *EDBT*, 20, pages 423–426, 2020. doi:10.5441/002/edbt.2020.48.

[30] L. Benson, H. Makait, and T. Rabl. **Viper: An efficient hybrid pmem-dram key-value store**. *Proceedings of the VLDB Endowment*, 14(9):1544–1556, 2021.

[31] L. Benson and T. Rabl. **Darwin: Scale-in stream processing**. In *CIDR*, 2022. https://www.cidrdb.org/cidr2022/papers/p34-benson.pdf.

[32] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. **IBM infosphere streams for scalable, real-time, intelligent transportation services**. In *SIGMOD*, pages 1093–1104. ACM, 2010. doi:10.1145/1807167.1807291.

[33] A. Bifet and R. Gavaldà. *Learning from Time-Changing Data with Adaptive Windowing*, pages 443–448. 2007. https://epubs.siam.org/doi/abs/10.1137/1.9781611972771.42, arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611972771.42, doi:10.1137/1.9781611972771.42.

[34] C. Binnig, R. Rehrmann, F. Faerber, and R. Riewe. **FunSQL: It is Time to Make SQL Functional**. In *DanaC*, page 41–46. ACM, 2012. doi:10.1145/2320765.2320786.

[35] I. Bird. **Computing for the large hadron collider**. *Annual Review of Nuclear and Particle Science*, 61:99–118, 2011.

[36] H.-J. Boehm, R. Atkinson, and M. Plass. **Ropes: an alternative to strings**. *Software: Practice and Experience*, 1995. doi:10.1002/spe.4380251203.

[37] M. Boehm, I. Antonov, S. Baunsgaard, M. Dokter, R. Ginthör, K. Innerebner, F. Klezin, S. N. Lindstaedt, A. Phani, B. Rath, B. Reinwald, S. Siddiqui, and S. B. Wrede. **SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle**. CIDR, 2019. http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf.

[38] M. Boehm, M. Interlandi, and C. Jermaine. **Optimizing Tensor Computations: From Applications to Compilation and Runtime Techniques**. In *Companion of the 2023 International Conference on Management of Data*, SIGMOD '23, page 53–59, New York, NY, USA, 2023. Association for Computing Machinery. https://doi.org/10.1145/3555041.3589407, doi:10.1145/3555041.3589407.

[39] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. **Tracing the meta-level: PyPy's tracing JIT compiler**. In *ICOOOLPS*, pages 18–25. ACM, 2009. doi:10.1145/1565824.1565827.

[40] P. A. Boncz, M. Zukowski, and N. Nes. **MonetDB/X100: Hyper-Pipelining Query Execution**. In *CIDR*, pages 225–237, 2005. http://cidrdb.org/cidr2005/papers/P19.pdf.

[41] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. J. Miller, and N. Tatbul. **SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems**. In *PVLDB*, 3, pages 232–243, 2010. doi:10.14778/1920841.1920874.

[42] J. Böttcher, V. Leis, J. Giceva, T. Neumann, and A. Kemper. **Scalable and robust latches for database systems**. In *DaMoN*. ACM, 2020. doi:10.1145/3399666.3399908.

[43] A. Brahmakshatriya and S. Amarasinghe. **BuildIt: A Type-Based Multi-stage Programming Framework for Code Generation in C++**. In *CGO*, 2021. doi:10.1109/CGO51591.2021.9370333.

[44] S. Bress, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. **Gpu-accelerated database systems: Survey and open challenges**. *Transactions on Large-Scale Data-and Knowledge-Centered Systems*, pages 1–35, 2014. doi:10.1007/978-3-662-45761-0_1.

[45] S. Bress, B. Köcher, H. Funke, S. Zeuch, T. Rabl, and V. Markl. **Generating custom code for efficient query execution on heterogeneous processors**. *The VLDB Journal*, 27:797–822, 2018. doi:10.1007/s00778-018-0512-y.

[46] Bress, Sebastian and Köcher, Bastian and Funke, Henning and Zeuch, Steffen and Rabl, Tilmann and Markl, Volker. **Generating Custom Code for Efficient Query Execution on Heterogeneous Processors**. *The VLDB Journal*, 27(6):797–822, 2018. doi:10.1007/s00778-018-0512-y.

[47] D. Broneske, S. Bress, and G. Saake. **Database scan variants on modern CPUs: A performance study**. In *IMDM*, pages 97–111. Springer, 2013. doi:10.1007/978-3-319-13960-9_8.

[48] D. Broneske, A. Meister, and G. Saake. **Hardware-sensitive scan operator variants for compiled selection pipelines**. In *BTW*, pages 403–412. Gesellschaft für Informatik, Bonn, 2017. https://dl.gi.de/20.500.12116/642.

[49] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos. **Beyond Analytics: The Evolution of Stream Processing Systems**. In *SIGMOD*, page 2651–2658. ACM, 2020. doi:10.1145/3318464.3383131.

[50] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. **Apache Flink: Stream and Batch Processing in a Single Engine**. *IEEE Data Engineering Bulletin*, 36(4), 2015. http://sites.computer.org/debull/A15dec/p28.pdf.

[51] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl. **Cutty: Aggregate sharing for user-defined windows**. In *CIKM*, pages 1201–1210, 2016. doi:10.1145/2983323.2983807.

[52] F. Carcillo, A. Dal Pozzolo, Y.-A. Le Borgne, O. Caelen, Y. Mazzer, and G. Bontempi. **Scarff: a scalable framework for streaming credit card fraud detection with spark**. *Information fusion*, 41:182–194, 2018.

[53] L. Cardelli and J. C. Mitchell. **Operations on records**. *Mathematical structures in computer science*, 1991. doi:10.1017/S0960129500000049.

[54] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. **Integrating scale out and fault tolerance in stream processing using operator state management**. In *SIGMOD*, pages 725–736. ACM, 2013.

[55] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, et al. **A history and evaluation of System R**. *Communications of the ACM*, 24(10):632–646, 1981. doi:10.1016/0166-5316(81)90053-5.

[56] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. **Trill: A High-performance Incremental Query Processor for Diverse Analytics**. In *PVLDB*, 8, pages 401–412. VLDB Endowment, 2014. doi:10.14778/2735496.2735503.

[57] B. Chandramouli, J. Goldstein, M. Barnett, and J. F. Terwilliger. **Trill: Engineering a Library for Diverse Analytics**. *IEEE Data Engineering Bulletin*, 38(4):51–60, 2015. http://sites.computer.org/debull/A15dec/p51.pdf.

[58] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. A. Shah. **TelegraphCQ: Continuous Dataflow Processing for an Uncertain World**. In *CIDR*, 2, page 4, 2003. http://www-db.cs.wisc.edu/cidr/cidr2003/program/p24.pdf.

[59] S. Chatterjee, M. Jagadeesan, W. Qin, and S. Idreos. **Cosine: A Cloud-Cost Optimized Self-Designing Key-Value Storage Engine**. *PVLDB*, 15(1):112–126, sep 2021. doi:10.14778/3485450.3485461.

[60] A. Chaudhary, S. Zeuch, V. Markl, and J. Karimov. **Incremental Stream Query Merging**. In *EDBT 2023*, pages 604–617. OpenProceedings.org, 2023. doi:10.48786/edbt.2023.51.

[61] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. **NiagaraCQ: A scalable continuous query system for internet databases**. In *ACM SIGMOD Record*, 29, pages 379–390. ACM, 2000. doi:10.1145/335191.335432.

[62] K. Cheng. **A Computed Column Defined with a User-Defined Function Might Impact Query Performance**, 2011. https://blogs.msdn.microsoft.com/sqlcat/2011/11/28/a-computed-column-defined-with-a-user-defined-function-might-impact-query-performance/.

[63] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. **StatusQuo: Making Familiar Abstractions Perform Using Program Analysis.** In *CIDR*, 2013.

[64] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers. **Using Program Analysis to Improve Database Applications.** *Data Engineering Bulletin*, 2014.

[65] A. Cheung, A. Solar-Lezama, and S. Madden. **Optimizing database-backed applications with query synthesis**. In *SIGPLAN Notices*. ACM, 2013. doi:10.1145/2491956.2462180.

[66] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. **One trillion edges: Graph processing at facebook-scale**. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[67] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. Peng, and P. Poulosky. **Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming**. In *IPDPS*, pages 1789–1792. IEEE, 2016. doi:10.1109/IPDPSW.2016.138.

[68] J. Cieslewicz and K. A. Ross. **Adaptive aggregation on chip multiprocessors**. In *VLDB*, pages 339–350. VLDB Endowment, 2007.

[69] I. Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 6 2016.

[70] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. **An Architecture for Compiling UDF-Centric Workflows**. *PVLDB*, 8(12):1466–1477, aug 2015. doi:10.14778/2824032.2824045.

[71] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Çetintemel, and S. B. Zdonik. **Tupleware: "Big" Data, Big Analytics, Small Clusters**. In *CIDR*, 2015. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper23u.pdf.

[72] A. Crotty, A. Galakatos, and T. Kraska. **Getting Swole: Generating Access-Aware Code with Predicate Pullups**. In *IEEE ICDE*, pages 1273–1284, 2020. doi:10.1109/ICDE48307.2020.00114.

[73] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. **Efficiently computing static single assignment form and the control dependence graph**. *TOPLAS*, 1991. doi:10.1145/115372.115320.

[74] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. **The Snowflake Elastic Data Warehouse**. In *ACM SIGMOD*, SIGMOD '16, page 215–226, New York, NY, USA, 2016. Association for Computing Machinery. https://doi.org/10.1145/2882903.2903741, doi:10.1145/2882903.2903741.

[75] P. Damme, M. Birkenbach, C. Bitsakos, M. Boehm, P. Bonnet, F. M. Ciorba, M. Dokter, P. Dowgiallo, A. Eleliemy, C. Faerber, G. I. Goumas, D. Habich, N. Hedam, M. Hofer, W. Huang, K. Innerebner, V. Karakostas, R. Kern, T. Kosar, A. Krause, D. Krems, A. Laber, W. Lehner, E. Mier, M. Paradies, B. Peischl, G. Poerwawinata, S. Psomadakis, T. Rabl, P. Ratuszniak, P. Silva, N. Skuppin, A. Starzacher, B. Steinwender, I. Tolovski, P. Tözün, W. Ulatowski, Y. Wang, I. P. Wrosz, A. Zamuda, C. Zhang, and X. Zhu. **DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines**. In *CIDR 2022*. www.cidrdb.org, 2022. https://www.cidrdb.org/cidr2022/papers/p4-damme.pdf.

[76] M. Dashti, S. B. John, T. Coppey, A. Shaikhha, V. Jovanovic, and C. Koch. **Compiling Database Application Programs**. *CIDR*, 2018.

[77] Decodable. **Decodable**, 2022. https://www.decodable.co/.

[78] B. Del Monte, S. Zeuch, T. Rabl, and V. Markl. **Rethinking Stateful Stream Processing with RDMA**. In *SIGMOD*, page 1078–1092, New York, NY, USA, 2022. Association for Computing Machinery. https://doi.org/10.1145/3514221.3517826, doi:10.1145/3514221.3517826.

[79] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. **Hekaton: SQL Server's Memory-optimized OLTP Engine**. In *SIGMOD*, pages 1243–1254. ACM, 2013. doi:10.1145/2463676.2463710.

[80] B. Dong, K. Wu, S. Byna, J. Liu, W. Zhao, and F. Rusu. **ArrayUDF: User-Defined Scientific Data Analysis on Arrays**. In *HPDC*. ACM, 2017.

[81] M. Dreseler, J. Kossmann, J. Frohnhofen, M. Uflacker, and H. Plattner. **Fused Table Scans: Combining AVX-512 and JIT to Double the Performance of Multi-Predicate Scans**. In *ICDEW*, pages 102–109. IEEE, April 2018. doi:10.1109/icdew.2018.00024.

[82] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. **An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler**. In *VMIL*. ACM, 2013. doi:10.1145/2542142.2542143.

[83] K. Dursun, C. Binnig, U. Cetintemel, G. Swart, and W. Gong. **A morsel-driven query execution engine for heterogeneous multi-cores**. *PVLDB*, 12(12):2218–2229, 2019. doi:10.14778/3352063.3352137.

[84] C. Duta and T. Grust. **Functional-Style SQL UDFs With a Capital 'F'**. In *SIGMOD*. ACM, 2020.

[85] C. Duta, D. Hirn, and T. Grust. **Compiling PL/SQL Away**. In *CIDR*, 2019.

[86] A. Dutt and J. R. Haritsa. **Plan Bouquets: Query Processing Without Selectivity Estimation**. In *SIGMOD*, pages 1039–1050, New York, NY, USA, 2014. ACM.

[87] A. Eisenberg and J. Melton. **SQL: 1999, formerly known as SQL3**. *ACM Sigmod record*, 28(1):131–138, 1999. doi:10.1145/309844.310075.

[88] K. V. Emani, T. Deshpande, K. Ramachandra, and S. Sudarshan. **DBridge: Translating Imperative Code to SQL**. In *SIGMOD*. ACM, 2017.

[89] K. V. Emani, K. Ramachandra, S. Bhattacharya, and S. Sudarshan. **Extracting Equivalent SQL from Imperative Code in Database Applications**. In *SIGMOD*. ACM, 2016.

[90] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf. **Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data**. In *OSDI*, pages 799–815, 2018. https://www.usenix.org/conference/osdi18/presentation/essertel.

[91] G. M. Essertel, R. Y. Tahboub, and T. Rompf. **On-stack replacement for program generators and source-to-source compilers**. In *GPCE*, pages 156–169. ACM, 2021. doi:10.1145/3486609.3487207.

[92] M. Faust, D. Schwalb, and J. Krueger. **Fast column scans: Paged indices for in-memory column stores**. In *IMDM*, pages 15–27. Springer, 2013. doi:10.1007/978-3-319-13960-9_2.

[93] W. Feng. **Four Billion Records per Second! What is Behind Alibaba Double 11 - Flink Stream-Batch Unification Practice during Double 11 for the Very First Time — alibabacloud.com**. https://www.alibabacloud.com/blog/four-billion-records-per-second-stream-batch-integration-implementation-of-alibaba-cloud-realtime-compute-for-apache-flink-during-double-11_596962, 2020. [Accessed 05-Jun-2023].

[94] C. D. Ferrara. **Light up the Spark in catalyst by avoiding UDF**. https://www.codemotion.com/magazine/light-up-the-spark-in-catalyst-by-avoiding-udf-4061.

[95] Y. Y. M. I. D. Fetterly, M. Budiu, Ú. Erlingsson, and P. K. G. J. Currey. **DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language**. *LSDS-IR*, 2009.

[96] S. J. Fink and F. Qian. **Design, implementation and evaluation of adaptive recompilation with on-stack replacement**. In *CGO*, pages 241–252. IEEE, 2003. doi:10.1109/CGO.2003.1191549.

[97] A. Fog. **How good is hyperthreading?** https://www.agner.org/optimize/blog/read.php?i=6, 2009. [Online; accessed 31.5.2019].

[98] P. S. Foundation. **CPython**. https://github.com/python/cpython, 2020.

[99] P. S. Foundation. **Duck-Typing**. https://docs.python.org/3/glossary.html#term-duck-typing, 2020.

[100] P. S. Foundation. **C-API**. https://docs.python.org/3/c-api/, 2021.

[101] T. A. S. Foundation. **Apache Arrow**. https://arrow.apache.org/, 2021.

[102] C. Freedman, E. Ismert, and P.-Å. Larson. **Compilation in the Microsoft SQL Server Hekaton Engine**. *IEEE Data Engineering Bulletin*, 37:22–30, 2014. http://sites.computer.org/debull/A14mar/p22.pdf.

[103] H. Funke, J. Mühlig, and J. Teubner. **Efficient Generation of Machine Code for Query Compilers**. In *DaMoN*. ACM, 2020. doi:10.1145/3399666.3399925.

[104] H. Funke and J. Teubner. **Data-parallel query processing on non-uniform data**. *PVLDB*, 13(6):884–897, 2020. doi:10.14778/3380750.3380758.

[105] Y. Futamura. **Partial evaluation of computation process-an approach to a compiler-compiler**. *Systems, computers, controls*, 1971. doi:10.1023/A:1010095604496.

[106] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. **Trace-based just-in-time type specialization for dynamic languages**. *PLDI*, pages 465–478, 2009. doi:10.1145/1542476.1542528.

[107] H. Garcia-Molina et al. **Database Systems—The Complete Book**. *Database systems the complete book,*, 2005.

[108] H. Gavriilidis. **Computation Offloading in JVM-based Dataflow Engines**. *BTW*, 2019.

[109] B. Gedik, H. Andrade, and K. Wu. **A code generation approach to optimizing high-performance distributed data stream processing**. In *CIKM*, pages 847–856. ACM, 2009. doi:10.1145/1645953.1646061.

[110] G. E. Gévay, J.-A. Quiané-Ruiz, and V. Markl. **The Power of Nested Parallelism in Big Data Processing – Hitting Three Flies with One Slap –**. In *SIGMOD*. ACM, 2021.

[111] G. E. Gévay, T. Rabl, S. Bress, L. Madai-Tahy, and V. Markl. **Labyrinth: Compiling imperative control flow to parallel dataflows**. *arXiv preprint arXiv:1809.06845*, 2018.

[112] G. E. Gévay, T. Rabl, S. Bress, L. Madai-Tahy, J.-A. Quiané-Ruiz, and V. Markl. **Efficient control flow in dataflow systems: When ease-of-use meets high performance**. In *ICDE*. IEEE, 2021.

[113] C. A. Gomez-Uribe and N. Hunt. **The netflix recommender system: Algorithms, business value, and innovation**. *TMIS*, 6(4):1–19, 2015. doi:10.1145/2843948.

[114] Google. **v8**. https://v8.dev/, 2020.

[115] Google. **How Google Search organizes information**. https://www.google.com/intl/en/search/howsearchworks/how-search-works/organizing-information/, 2023. Accessed: 2023-06-26.

[116] G. Graefe. **Encapsulation of Parallelism in the Volcano Query Processing System**. In *SIGMOD*, pages 102–111. ACM, 1990. doi:10.1145/93605.98720.

[117] G. Graefe. **Volcano/spl minus/an extensible and parallel query evaluation system**. *TKDE*, 1994.

[118] J. Grier. **Extending the Yahoo! Streaming Benchmark**. https://data-artisans.com/blog/extending-the-yahoo-streaming-benchmark, 2016. [Online; accessed 31.5.2019].

[119] Y. Grigorov, H. Gavriilidis, S. Redyuk, K. Beedkar, and V. Markl. **P2D: A Transpiler Framework for Optimizing Data Science Pipelines**. DEEM '23. ACM, 2023. doi:10.1145/3595360.3595853.

[120] M. Grimmer, C. Seaton, R. Schatz, T. Würthinger, and H. Mössenböck. **High-performance cross-language interoperability in a multi-language runtime**. In *Symposium on Dynamic Languages*, 2015.

[121] F. Gruber, M. Bandle, A. Engelke, T. Neumann, and J. Giceva. **Bringing Compiling Databases to RISC Architectures**. *PVLDB*, 16(6):1222–1234, apr 2023. https://doi.org/10.14778/3583140.3583142, doi:10.14778/3583140.3583142.

[122] P. M. Grulich, A. Lepping, D. P. A. Nugroho, B. Del Monte, V. Pandey, S. Zeuch, and V. Markl. **Towards unifying query interpretation and compilation**. CIDR, 2023.

[123] P. M. Grulich and F. Nawab. **Collaborative Edge and Cloud Neural Networks for Real-Time Video Processing**. *Proc. VLDB Endow.*, 11(12):2046–2049, aug 2018. https://doi.org/10.14778/3229863.3236256, doi:10.14778/3229863.3236256.

[124] P. M. Grulich, R. Saitenmacher, J. Traub, S. Bress, T. Rabl, and V. Markl. **Scalable Detection of Concept Drifts on Data Streams with Parallel Adaptive Windowing**. In *EDBT*, pages 477–480, 2018. doi:10.5441/002/edbt.2018.51.

[125] P. M. Grulich, B. Sebastian, S. Zeuch, J. Traub, J. v. Bleichert, Z. Chen, T. Rabl, and V. Markl. **Grizzly: Efficient Stream Processing Through Adaptive Query Compilation**. In *SIGMOD*, page 2487–2503. ACM, 2020. doi:10.1145/3318464.3389739.

[126] P. M. Grulich, J. Traub, S. Bress, A. Katsifodimos, V. Markl, and T. Rabl. **Generating reproducible out-of-order data streams**. In *DEBS*, pages 256–257. ACM, 2019. doi:10.1145/3328905.3332511.

[127] P. M. Grulich, S. Zeuch, and V. Markl. **Towards Efficient and Secure UDF Execution with BabelfishLib (Lightning Talk)**. In *VLDB 2023*, 3462 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2023.

[128] P. M. Grulich, S. Zeuch, and V. Markl. **Babelfish: Efficient Execution of Polyglot Queries**. *Proc. VLDB Endow.*, 15(2):196–210, oct 2021. https://doi.org/10.14778/3489496.3489501, doi:10.14778/3489496.3489501.

[129] T. Gubner and P. Boncz. **Charting the Design Space of Query Execution Using VOILA**. *PVLDB*, 14(6):1067–1079, feb 2021. https://doi.org/10.14778/3447689.3447709, doi:10.14778/3447689.3447709.

[130] T. Gubner and P. Boncz. **Excalibur: A Virtual Machine for Adaptive Fine-grained JIT-Compiled Query Execution based on VOILA**. *PVLDB*, 16(4):829–841, 2022. doi:10.14778/3574245.3574266.

[131] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. **Spotting code optimizations in data-parallel pipelines through periscope**. In *OSDI*, 2012.

[132] S. Gupta, S. Purandare, and K. Ramachandra. **Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates**. In *SIGMOD*, 2020.

[133] I. Haffner and J. Dittrich. **A Simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries**. In *EDBT 2023*. OpenProceedings.org, 2023. doi:10.48786/edbt.2023.01.

[134] S. Hagedorn, S. Kläbe, and K.-U. Sattler. **Putting Pandas in a Box.** In *CIDR*, 2021.

[135] C. Heinzelman, 2011. https://blogs.msdn.microsoft.com/sqlcat/2011/06/24/unintended-consequences-of-scalar-valued-user-defined-functions/.

[136] A. Heise, A. Rheinländer, M. Leich, U. Leser, and F. Naumann. **Meteor/sopremo: An extensible query language and operator model**. In *BigData*. Citeseer, 2012.

[137] T. Hey, S. Tansley, K. Tolle, and J. Gray. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, October 2009. https://www.microsoft.com/en-us/research/publication/fourth-paradigm-data-intensive-scientific-discovery/.

[138] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. **A Catalog of Stream Processing Optimizations**. *ACM Computing Surveys*, 46(4), March 2014. https://doi.org/10.1145/2528412.

[139] U. Hölzle, C. Chambers, and D. Ungar. **Debugging optimized code with dynamic deoptimization**. In *ACM Sigplan Notices*, 27, pages 32–43. ACM, 1992. doi:10.1145/143095.143114.

[140] C. Hypervisor. **Cloud Hypervisor**, 2023. [Online; accessed 06.06.2023]. https://www.cloudhypervisor.org/.

[141] IBM. **Avoid UDFs as join predicates**. https://www.ibm.com/support/knowledgecenter/en/SSPT3X_4.2.0/com.ibm.swg.im.infosphere.biginsights.text.doc/doc/ana_txtan_udf-join-guideline.html, 2020.

[142] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. **Here are my data files. here are my queries. where are my results?** In *CIDR*, 2011.

[143] S. Idreos, M. L. Kersten, and S. Manegold. **Database Cracking**. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 68–78. www.cidrdb.org, 2007. http://cidrdb.org/cidr2007/papers/cidr07p07.pdf.

[144] Intel. **Intel(R) Threading Building Blocks: Concurrent Hash Map**. https://software.intel.com/en-us/node/506191, 2019. [Online; accessed 31.5.2019].

[145] K. Ishizaki. **Analyzing and Optimizing Java Code Generation for Apache Spark Query Plan**. In *ICPE*. ACM, 2019.

[146] G. Jacques-Silva, R. Lei, L. Cheng, G. J. Chen, K. Ching, T. Hu, Y. Mei, K. Wilfong, R. Shetty, S. Yilmaz, et al. **Providing streaming joins as a service at facebook**. *Proceedings of the VLDB Endowment*, 11(12):1809–1821, 2018.

[147] A. Jayarajan, W. Zhao, Y. Sun, and G. Pekhimenko. **TiLT: A Time-Centric Approach for Stream Query Optimization and Parallelization**. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 818–832, New York, NY, USA, 2023. ACM. doi:10.1145/3575693.3575704.

[148] P. Jesus, C. Baquero, and P. S. Almeida. **A survey of distributed data aggregation algorithms**. *IEEE Communications Surveys & Tutorials*, 17(1):381–404, 2014. doi:10.1109/COMST.2014.2354398.

[149] A. Jindal, K. V. Emani, M. Daum, O. Poppe, B. Haynes, A. Pavlenko, A. Gupta, K. Ramachandra, C. Curino, A. Mueller, et al. **Magpie: Python at speed and scale using cloud backends**. In *CIDR*, 2021.

[150] N. D. Jones. **An Introduction to Partial Evaluation**. *ACM Computing Surveys*, 1996. doi:10.1145/243439.243447.

[151] M. Jungmair, A. Kohn, and J. Giceva. **Designing an Open Framework for Query Optimization and Compilation**. *PVLDB*, 15(11):2389–2401, jul 2022. doi:10.14778/3551793.3551801.

[152] K. Karanasos, M. Interlandi, D. Xin, F. Psallidas, R. Sen, K. Park, I. Popivanov, S. Nakandal, S. Krishnan, M. Weimer, et al. **Extending relational query processing with ML inference**. In *CIDR*, 2020.

[153] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. **Benchmarking distributed stream data processing systems**. In *ICDE*, pages 1507–1518. IEEE, 2018. doi:10.1109/ICDE.2018.00169.

[154] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. **Fast Queries over Heterogeneous Data through Engine Customization**. *PVLDB*, 2016. doi:10.14778/2994509.2994516.

[155] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. **Adaptive Query Processing on RAW Data**. *PVLDB*, 7(12):1119–1130, aug 2014. https://doi.org/10.14778/2732977.2732986, doi:10.14778/2732977.2732986.

[156] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. A. Boncz. **Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask**. *PVLDB*, 2018. doi:10.14778/3275366.3275370.

[157] T. Kersten, V. Leis, and T. Neumann. **Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra**. *VLDB J.*, 2021. doi:10.1007/s00778-020-00643-4.

[158] T. Kiefer, B. Schlegel, and W. Lehner. **Experimental evaluation of NUMA effects on database management systems**. In *BTW*. Gesellschaft für Informatik eV, 2013.

[159] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. **The Tensor Algebra Compiler**. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. https://doi.org/10.1145/3133901, doi:10.1145/3133901.

[160] S. Kläbe, R. DeSantis, S. Hagedorn, and K. Sattler. **Accelerating Python UDFs in Vectorized Query Execution**. In *CIDR*, 2022. https://www.cidrdb.org/cidr2022/papers/p33-klaebe.pdf.

[161] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. **Building efficient query engines in a high-level language**. In *PVLDB*, 7, pages 853–864. VLDB Endowment, 2014. doi:10.14778/2732951.2732959.

[162] A. Kohn, V. Leis, and T. Neumann. **Adaptive execution of compiled queries**. In *ICDE*, pages 197–208. IEEE, 2018.

[163] A. Kohn, V. Leis, and T. Neumann. **Building Advanced SQL Analytics From Low-Level Plan Operators**. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1001–1013. ACM, 2021. https://doi.org/10.1145/3448016.3457288, doi:10.1145/3448016.3457288.

[164] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. **Saber: Window-based hybrid stream processing for heterogeneous architectures**. In *SIGMOD*, pages 555–569. ACM, 2016. doi:10.1145/2882903.2882906.

[165] H. Kornelis. **T-SQL User-Defined Functions: the good, the bad, and the ugly**, 2012. https://sqlserverfast.com/blog/hugo/2012/05/t-sql-user-defined-functions-the-good-the-bad-and-the-ugly-part-1/.

[166] K. Krikellas, S. D. Viglas, and M. Cintra. **Generating code for holistic query evaluation**. In *ICDE*, pages 613–624, 2010.

[167] L. Kroll, K. Segeljakt, P. Carbone, C. Schulte, and S. Haridi. **Arc: An IR for Batch and Stream Programming**. In *DBPL*, pages 53–58, New York, NY, USA, 2019. ACM. doi:10.1145/3315507.3330199.

[168] A. Kunft, A. Katsifodimos, S. Schelter, S. Bress, T. Rabl, and V. Markl. **An intermediate representation for optimizing machine learning pipelines**. *PVLDB*, 2019.

[169] A. Kunft, A. Katsifodimos, S. Schelter, T. Rabl, and V. Markl. **Blockjoin: Efficient Matrix Partitioning through Joins**. *PVLDB*, 10(13):2061–2072, sep 2017. doi:10.14778/3151106.3151110.

[170] A. Kunft, L. Stadler, D. Bonetta, C. Basca, J. Meiners, S. Bress, T. Rabl, J. J. Fumero, and V. Markl. **ScootR: Scaling R Dataframes on Dataflow Systems.** In *SoCC*. ACM, 2018.

[171] S. K. Lam, A. Pitrou, and S. Seibert. **Numba: a LLVM-based Python JIT compiler**. In *Workshop on the LLVM Compiler Infrastructure in HPC*, pages 1–6. ACM, 2015. doi:10.1145/2833157.2833162.

[172] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. **Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation**. In *Proceedings of the 2016 International Conference on Management of Data*, pages 311–326, 2016.

[173] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, and M. Hoffmann. **Industry 4.0**. *Business & information systems engineering*, 6:239–242, 2014.

[174] J. Laskowski. **UDFs are Blackbox-Don't Use Them Unless You've Got No Choice**. https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-udfs-blackbox.html, 2021.

[175] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. **MLIR: Scaling Compiler Infrastructure for Domain Specific Computation**. In *CGO*. IEEE, 2021. doi:10.1109/cgo51591.2021.9370308.

[176] V. Leis, P. Boncz, A. Kemper, and T. Neumann. **Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age**. In *SIGMOD*, pages 743–754. ACM, 2014. doi:10.1145/2588555.2610507.

[177] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. **LeanStore: In-memory data management beyond main memory**. In *ICDE*, pages 185–196. IEEE, 2018. doi:10.1109/icde.2018.00026.

[178] V. Leis, A. Kemper, and T. Neumann. **The adaptive radix tree: ARTful indexing for main-memory databases**. In *ICDE*, pages 38–49. IEEE, 2013. doi:10.1109/ICDE.2013.6544812.

[179] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. **Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark**. *The VLDB Journal*, 27(5):643–668, October 2018.

[180] D. Leopoldseder, L. Stadler, M. Rigger, T. Würthinger, and H. Mössenböck. **A Cost Model for a Graph-based Intermediate-representation in a Dynamic Compiler**. In *VMIL*, 2018. doi:10.1145/3281287.3281290.

[181] A. Lepping, H. M. Pham, L. Mons, B. Rueb, A. Chaudhary, P. M. Grulich, S. Zeuch, and V. Markl. **Showcasing Data Management Challenges for Future IoT Applications with NebulaStream**. *VLDB*, 2023.

[182] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. M. Lohman. **NUMA-aware algorithms: the case of data shuffling.** In *CIDR*, 2013.

[183] S. Liang. *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional, 1999.

[184] G. M. Lohman. **Grammar-like Functional Rules for Representing Query Optimization Alternatives**. In H. Boral and P. Larson, editors, *SIGMOD*, pages 18–27. ACM, 1988. doi:10.1145/50202.50204.

[185] C. Lutz, S. Bress, S. Zeuch, T. Rabl, and V. Markl. **Pump up the volume: Processing large data on GPUs with fast interconnects**. In *SIGMOD*, pages 1633–1649. ACM, 2020. doi:10.1145/3318464.3389705.

[186] H. Mai, B. Liu, and N. Cherukuri. **Introducing AthenaX, Uber engineering's open source streaming analytics platform**, 2017.

[187] T. Maier, P. Sanders, and R. Dementiev. **Concurrent Hash Tables: Fast and General (?)!** *TOPC*, 5(4):16, 2019. doi:10.1145/3016078.2851188.

[188] V. Makarov. **MIR: A lightweight JIT compiler project**. https://developers.redhat.com/blog/2020/01/20/mir-a-lightweight-jit-compiler-project, 2020. [Online; accessed 22.6.2023].

[189] S. Mandl, O. Kozachuk, and J. Graupmann. **Bring Your Language to Your Data with EXASOL**. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)*, 2017.

[190] S. Manegold, M. L. Kersten, and P. Boncz. **Database architecture evolution: Mammals flourished long before dinosaurs became extinct**. *Proceedings of the VLDB Endowment*, 2(2):1648–1653, 2009.

[191] W. McKinney. **Data Structures for Statistical Computing in Python**. In *SciPy*. scipy.org, 2010. doi:10.25080/Majora-92bf1922-00a.

[192] F. McSherry, M. Isard, and D. G. Murray. **Scalability! but at what cost?** In *HotOS*, 15, pages 14–14, 2015.

[193] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar. **MLlib: Machine Learning in Apache Spark**. *The Journal of Machine Learning Research*, 2016. http://jmlr.org/papers/v17/15-237.html.

[194] P. Menon, T. C. Mowry, and A. Pavlo. **Relaxed Operator Fusion for In-memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last**. In *PVLDB*, 11, pages 1–13. VLDB Endowment, 2017. doi:10.14778/3151113.3151114.

[195] P. Menon, A. Ngom, L. Ma, T. C. Mowry, and A. Pavlo. **Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling**. *PVLDB*, 2020. doi:10.14778/3425879.3425882.

[196] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. **StreamBox: Modern Stream Processing on a Multicore Machine**. In *USENIX*, pages 617–629, 2017. https://www.usenix.org/conference/atc17/technical-sessions/presentation/miao.

[197] A. Michalke, P. M. Grulich, C. Lutz, S. Zeuch, and V. Markl. **An energy-efficient stream join for the Internet of Things**. In *DaMoN*, pages 1–6, 2021. doi:10.1145/3465998.3466005.

[198] J. Mintz. **In this iteration of Database Deep Dives, we had the pleasure of catching up with Professor Andy Pavlo**, 2017. https://www.ibm.com/cloud/blog/database-deep-dives-with-andy-pavlo.

[199] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Kemper, and T. Neumann. **Heterogeneity-Conscious Parallel Query Execution: Getting a better mileage while driving faster!** In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, pages 1–10, 2014.

[200] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. **Naiad: a timely dataflow system**. In *SOSP*, pages 439–455. ACM, 2013. doi:10.1145/2517349.2522738.

[201] C. Navasca, C. Cai, K. Nguyen, B. Demsky, S. Lu, M. Kim, and G. H. Xu. **Gerenuk: Thin Computation over Big Native Data Using Speculative Program Transformation**. In *SOSP*. ACM, 2019.

[202] T. Neumann. **Efficiently Compiling Efficient Query Plans for Modern Hardware**. In *PVLDB*, 4, pages 539–550. VLDB Endowment, 2011.

[203] T. Neumann and M. J. Freitag. **Umbra: A Disk-Based System with In-Memory Performance**. In *CIDR*, 2020. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf.

[204] T. Neumann and G. Moerkotte. **Generating optimal DAG-structured query evaluation plans**. *Computer Science-Research and Development*, 2009. doi:10.1007/s00450-009-0061-0.

[205] T. Neumann, T. Mühlbauer, and A. Kemper. **Fast serializable multi-version concurrency control for main-memory database systems**. In *SIGMOD*, pages 677–689. ACM, 2015. doi:10.1145/2723372.2749436.

[206] D. Nugroho, P. M. Grulich, C. Lutz, S. Bortoli, and V. Zeuch, Steffen Markl. **Benchmarking Stream Join Algorithms on GPUs: A Framework and its Application to the State-of-the-art**. In *EDBT 2024*, 2024.

[207] B. of Transportation Statistics. **Reporting Carrier On-Time Performance**. https://www.transtats.bts.gov/Tables.asp?DB_ID=120, 2020. [Online; accessed 22.2.2021].

[208] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. **Pig latin: a not-so-foreign language for data processing**. In *SIGMOD*. ACM, 2008.

[209] Oracle. **Graal Native Image**. https://www.graalvm.org/reference-manual/native-image/, 2020.

[210] Oracle. **Graal Python**. https://github.com/graalvm/graalpython, 2020.

[211] Oracle. **GraalJS**. https://github.com/graalvm/graaljs, 2020.

[212] B. Ozar. **Froid: How SQL Server 2019 Will Fix the Scalar Functions Problem**, 2019. https://www.brentozar.com/archive/2018/01/froid-sql-server-vnext-might-fix-scalar-functions-problem/.

[213] M. Paleczny, C. Vick, and C. Click. **The java hotspot TM server compiler**. In *JVM*. USENIX, 2001.

[214] S. Palkar, J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. P. Amarasinghe, S. Madden, and M. Zaharia. **Evaluating End-to-End Optimization for Data Analytics Applications in Weld**. *PVLDB*, 11(9):1002–1015, 2018. doi:10.14778/3213880.3213890.

[215] S. Palkar, J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. P. Amarasinghe, and M. Zaharia. **Weld: A common runtime for high performance data analytics**. In *CIDR*, 2017. http://cidrdb.org/cidr2017/papers/p127-palkar-cidr17.pdf.

[216] P. Paroski. **Code generation: The inner sanctum of database performance**. http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-ofdatabase-performance.html, 2016. [Online; accessed 31.5.2019].

[217] P. Paroski. **Code generation: The inner sanctum of database performance**. http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-ofdatabase-performance.html, 2016. [Online; accessed 31.5.2019].

[218] M. Pasumansky and B. Wagner. **Assembling a Query Engine From Spare Parts**. In *CDMS*, 2022. https://cdmsworkshop.github.io/2022/Proceedings/ShortPapers/Paper1_MoshaPasumansky.pdf.

[219] J. Paul, B. He, S. Lu, and C. T. Lau. **Improving execution efficiency of just-in-time compilation based query processing on GPUs**. *PVLDB*, 14(2):202–214, 2020. doi:10.14778/3425879.3425890.

[220] G. Paulley. http://glennpaulley.ca/database/2015/07/performance-overhead-of-sql-user-defined-functions/.

[221] P. Pedreira, O. Erling, M. Basmanova, K. Wilfong, L. S. Sakka, K. Pai, W. He, and B. Chattopadhyay. **Velox: Meta's Unified Execution Engine**. *PVLDB*, 15(12):3372–3384, 2022. doi:10.14778/3554821.3554829.

[222] P. Pietzuch, P. Garefalakis, A. Koliousis, H. Pirk, and G. Theodorakis. **Do We Need Distributed Stream Processing?** https://lsds.doc.ic.ac.uk/blog/do-we-need-distributed-stream-processing, 2018. [Online; accessed 31.5.2019].

[223] P. Pietzuch, P. Garefalakis, A. Koliousis, H. Pirk, and G. Theodorakis. **StreamBench**. https://github.com/lsds/StreamBench, 2018. [Online; accessed 31.5.2019].

[224] H. Pirk, J. Giceva, and P. R. Pietzuch. **Thriving in the No Man's Land between Compilers and Databases**. In *CIDR*, 2019. http://cidrdb.org/cidr2019/papers/p91-pirk-cidr19.pdf.

[225] H. Pirk, O. Moll, M. Zaharia, and S. Madden. **Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware**. In *PVLDB*, 9, pages 1707–1718. VLDB Endowment, 2016. doi:10.14778/3007328.3007336.

[226] PostgreSQL. **PostgreSQL**. https://www.postgresql.org/, 2020.

[227] V. Prajapati. *Big data analytics with R and Hadoop*. Packt Publishing Ltd, 2013.

[228] F. Qaiser. **UDFs vs. Map vs. Custom Spark-Native Functions**, 2018. https://medium.com/@Farox2q/udfs-vs-map-vs-custom-spark-native-functions-91ab2c154b44.

[229] M. Raasveldt. **Push-Based Execution in DuckDB**, 2022. https://dsdsd.da.cwi.nl/slides/dsdsd-duckdb-push-based-execution.pdf.

[230] M. Raasveldt and H. Mühleisen. **Vectorized UDFs in Column-Stores**. In *SSDBM*. ACM, 2016.

[231] M. Raasveldt and H. Mühleisen. **DuckDB: an embeddable analytical database**. In *SIGMOD*, pages 1981–1984. ACM, 2019. doi:10.1145/3299869.3320212.

[232] B. Răducanu, P. Boncz, and M. Zukowski. **Micro adaptivity in vectorwise**. In *SIGMOD*, pages 1231–1242. ACM, 2013. doi:10.1145/2588555.2588566.

[233] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham. **Froid: Optimization of Imperative Programs in a Relational Database**. *PVLDB*, 11(4):432–444, dec 2017. https://doi.org/10.1145/3186728.3164140, doi:10.1145/3186728.3164140.

[234] J. Rao, H. Pirahesh, C. Mohan, and G. Lohman. **Compiled query execution engine using JVM**. In *ICDE*. IEEE, 2006. doi:10.1109/ICDE.2006.40.

[235] G. Renoux and K. Kontoudi. **Real-time bot mitigation with machine learning in Flink**, 2021.

[236] A. Rheinländer, U. Leser, and G. Graefe. **Optimization of Complex Dataflows with User-Defined Functions**. *ACM Computing Surveys*, 2017.

[237] T. Rompf and M. Odersky. **Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs**. In *Proceedings of the ninth international conference on Generative programming and component engineering*, pages 127–136, 2010.

[238] V. Rosenfeld, R. Mueller, P. Tözün, and F. Özcan. **Processing Java UDFs in a C++ Environment**. In *SoCC*. ACM, 2017.

[239] K. A. Ross. **Conjunctive Selection Conditions in Main Memory**. In *SIGMOD*. ACM, 2002. doi:10.1145/543613.543628.

[240] K. A. Ross. **Selection conditions in main memory**. *TODS*, 29(1):132–161, 2004. doi:10.1145/974750.974755.

[241] K. Saur, T. Mirmira, K. Karanasos, and J. Camacho-Rodríguez. **Containerized execution of UDFs: an experimental evaluation**. *Proceedings of the VLDB Endowment*, 15(11):3158–3171, 2022.

[242] F. Schiavio, D. Bonetta, and W. Binder. **Towards Dynamic SQL Compilation in Apache Spark**. In *‹Programming›*, New York, NY, USA, 2020. ACM.

[243] F. Schiavio, D. Bonetta, and W. Binder. **Language-Agnostic Integrated Queries in a Managed Polyglot Runtime**. *PVLDB*, 14(8):1414–1426, 2021. doi:10.14778/3457390.3457405.

[244] F. Schiavio, D. Bonetta, and W. Binder. **DynQ: a dynamic query engine with query-reuse capabilities embedded in a polyglot runtime**. *The VLDB Journal*, pages 1–25, 03 2023. doi:10.1007/s00778-023-00784-2.

[245] T. Schmidt, P. Fent, and T. Neumann. **Efficiently Compiling Dynamic Code for Adaptive Query Processing**. In *ADMS*, 2022.

[246] N. Schubert, P. M. Grulich, S. Zeuch, and V. Markl. **Exploiting Access Pattern Characteristics for Join Reordering**. In *DaMoN 2023*, 2023. doi:10.1145/3592980.3595304.

[247] N. L. Schubert, P. M. Grulich, S. Zeuch, B. Del Monte, , and V. Markl. **Architecting a Dynamic and Efficient Stream Processing Engine**. In *Under Submission*.

[248] M. E. Schüle, J. Huber, A. Kemper, and T. Neumann. **Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL**. In *SSDBM*. ACM, 2020.

[249] A. Shaikhha, M. Dashti, and C. Koch. **Push versus pull-based loop fusion in query engines**. *Journal of Functional Programming*, 28:e10, 2018.

[250] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. **How to architect a query compiler**. In *SIGMOD*, 2016. doi:10.1145/2882903.2915244.

[251] M. Sichert and T. Neumann. **User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases**. *PVLDB*, 15(5):1119–1131, 2022.

[252] V. Simhadri, K. Ramachandra, A. Chaitanya, R. Guravannavar, and S. Sudarshan. **Decorrelation of user defined function invocations in queries**. In *ICDE*, pages 532–543. IEEE, 2014.

[253] L. Spiegelberg, R. Yesantharao, M. Schwarzkopf, and T. Kraska. **Tuplex: Data Science in Python at Native Code Speed**. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1718–1731, New York, NY, USA, 2021. Association for Computing Machinery. https://doi.org/10.1145/3448016.3457244, doi:10.1145/3448016.3457244.

[254] Spotify. **Product Lessons from ML Home: Spotify's One-Stop Shop for Machine Learning**. https://engineering.atspotify.com/2022/01/product-lessons-from-ml-home-spotifys-one-stop-shop-for-machine-learning/, 2022. Accessed: 2023-06-26.

[255] Stackoverflow. **Spark functions vs UDF performance?**, 2016. https://stackoverflow.com/questions/38296609/spark-functions-vs-udf-performance.

[256] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. **LEO-DB2's learning optimizer**. In *PVLDB*, 1, pages 19–28, 2001. http://www.vldb.org/conf/2001/P019.pdf.

[257] V. Stinner. **Python performance Past, Present, Future**. In *EuroPython*, 2019.

[258] J. Sun and M. Sfikas. **PyFlink: The integration of Pandas into PyFlink**. https://flink.apache.org/2020/08/04/pyflink-pandas-udf-support-flink.html, 2021.

[259] R. Y. Tahboub, G. M. Essertel, and T. Rompf. **How to architect a query compiler, revisited**. In *SIGMOD*, pages 307–322. ACM, 2018. doi:10.1145/3183713.3196893.

[260] R. Y. Tahboub and T. Rompf. **Architecting a Query Compiler for Spatial Workloads**. In *SIGMOD*, pages 2103–2118. ACM, 2020. doi:10.1145/3318464.3389701.

[261] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. **General incremental sliding-window aggregation**. In *PVLDB*, 8, pages 702–713. VLDB Endowment, 02 2015. doi:10.14778/2752939.2752940.

[262] P. Team. **PyPy**. https://foss.heptapod.net/pypy/pypy, 2020.

[263] T. H. Team. **HPy: a better API for Python**. https://github.com/hpyproject/hpy, 2021.

[264] D. Terpstra, H. Jagode, H. You, and J. Dongarra. **Collecting performance data with PAPI-C**. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010. doi:10.1007/978-3-642-11261-4_11.

[265] G. Theodorakis, A. Koliousis, P. Pietzuch, and H. Pirk. **LightSaber: Efficient Window Aggregation on Multi-Core Processors**. In *SIGMOD*, page 2505–2521. ACM, 2020. doi:10.1145/3318464.3389753.

[266] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. V. Ryaboy. **Storm@twitter**. In *SIGMOD*, pages 147–156. ACM, 2014. doi:10.1145/2588555.2595641.

[267] J. Traub, P. Grulich, A. R. Cuéllar, S. Bress, A. Katsifodimos, T. Rabl, and V. Markl. **Scotty: Efficient Window Aggregation for out-of-order Stream Processing**. In *ICDE*, pages 1300–1303. IEEE, 2018. doi:10.1109/ICDE.2018.00135.

[268] J. Traub, P. M. Grulich, A. R. Cuéllar, S. Bress, A. Katsifodimos, T. Rabl, and V. Markl. **Efficient Window Aggregation with General Stream Slicing**. In *EDBT*, pages 97–108, 2019. doi:10.5441/002/edbt.2019.10.

[269] J. Traub, P. M. Grulich, A. R. Cuellar, S. Bress, A. Katsifodimos, T. Rabl, and V. Markl. **Scotty: General and efficient open-source window aggregation for stream processing systems**. *TODS*, 46(1):1–46, 2021. doi:10.1145/3433675.

[270] J. Traub, N. Steenbergen, P. M. Grulich, T. Rabl, and V. Markl. **I2: Interactive Real-Time Visualization for Streaming Data.** In *EDBT*, pages 526–529, 2017.

[271] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. **Nexmark-a benchmark for queries over data streams**. Technical report, Technical Report. Technical report, OGI School of Science & Engineering at ..., 2008. https://datalab.cs.pdx.edu/niagara/pstream/nexmark.pdf.

[272] S. Vaarala. **Duktape**. https://duktape.org/, 2020.

[273] S. Venkataraman, Z. Yang, D. Liu, E. Liang, H. Falaki, X. Meng, R. Xin, A. Ghodsi, M. J. Franklin, I. Stoica, and M. Zaharia. **SparkR: Scaling R Programs with Spark**. In *SIGMOD*, pages 1099–1104. ACM, 2016. doi:10.1145/2882903.2903740.

[274] J. Verwiebe, P. M. Grulich, J. Traub, and V. Markl. **Algorithms for Windowed Aggregations and Joins on Distributed Stream Processing Systems**. *Datenbank-Spektrum*, 22(2):99–107, 2022. doi:10.1007/s13222-022-00417-y.

[275] J. Verwiebe, P. M. Grulich, J. Traub, and V. Markl. **Survey of window types for aggregation in stream processing systems**. *The VLDB Journal*, pages 1–27, 2023. doi:10.1007/s00778-022-00778-6.

[276] T. Vincenty. **Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations**. *Survey review*, 23(176):88–93, 1975. doi:10.1179/sre.1975.23.176.88.

[277] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Muehlbauer, T. Neumann, and M. Then. **Get real: How benchmarks fail to represent the real world**. In *DBTest*, 2018. doi:10.1145/3209950.3209952.

[278] S. Wanderman-Milne and N. Li. **Runtime Code Generation in Cloudera Impala**. *IEEE Data Engineering Bulletin*, 2014. http://sites.computer.org/debull/A14mar/p31.pdf.

[279] H. S. Warren. *Hacker's delight*. Pearson Education, 2013.

[280] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. **SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units**. *PVLDB*, 2(1):385–394, 2009. doi:10.14778/1687627.1687671.

[281] C. Wimmer and T. Würthinger. **Truffle: A Self-Optimizing Runtime System**. In *SPLASH*. ACM, 2012. doi:10.1145/2384716.2384723.

[282] Y. Wu and K.-L. Tan. **ChronoStream: Elastic stateful stream computation in the cloud**. In *ICDE*, pages 723–734. IEEE, 2015. doi:10.1109/ICDE.2015.7113328.

[283] T. Würthinger, C. Wimmer, C. Humer, A. Wöss, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. **Practical partial evaluation for high-performance dynamic language runtimes**. In *ACM SIGPLAN*, 2017. doi:10.1145/3062341.3062381.

[284] W. Yue, L. Benson, and T. Rabl. **Desis: Efficient Window Aggregation in Decentralized Networks**. EDBT, 2023.

[285] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. **Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing**. In *NSDI*. USENIX, 2012. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[286] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. **Discretized streams: Fault-tolerant streaming computation at scale**. In *SOSP*, pages 423–438. ACM, 2013. doi:10.21236/ada575859.

[287] S. Zeuch, S. Bress, T. Rabl, B. D. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, and V. Markl. **Analyzing Efficient Stream Processing on Modern Hardware**. *PVLDB*, 12(5):516–530, 2019. doi:10.14778/3303753.3303758.

[288] S. Zeuch, X. Chatziliadis, A. Chaudhary, D. Giouroukis, P. M. Grulich, D. P. A. Nugroho, A. Ziehn, and V. Mark. **NebulaStream: Data Management for the Internet of Things**. *Datenbank-Spektrum*, 22(2):131–141, 2022. doi:10.1007/s13222-022-00415-0.

[289] S. Zeuch, A. Chaudhary, B. D. Monte, H. Gavriilidis, D. Giouroukis, P. M. Grulich, S. Bress, J. Traub, and V. Markl. **The NebulaStream Platform for Data and Application Management in the Internet of Things**. In *CIDR*, 2020. http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf.

[290] S. Zeuch and J.-C. Freytag. **QTM: modelling query execution with tasks**. In *ADMS*, pages 34–45, 2014. http://www.adms-conf.org/2014/adms14_zeuch.pdf.

[291] S. Zeuch and J.-C. Freytag. **Selection on modern CPUs**. In *ADMS*, page 5. ACM, 2015. doi:10.1145/2803140.2803145.

[292] S. Zeuch, H. Pirk, and J. Freytag. **Non-Invasive Progressive Optimization for In-Memory Databases**. *PVLDB*, 9(14):1659–1670, 2016. doi:10.14778/3007328.3007332.

[293] S. Zeuch, E. T. Zacharatou, S. Zhang, X. Chatziliadis, A. Chaudhary, B. Del Monte, D. Giouroukis, P. M. Grulich, A. Ziehn, and V. Mark. **Nebulastream: Complex analytics beyond the cloud**. *Open Journal of Internet Of Things (OJIOT)*, 6(1):66–81, 2020. https://www.ronpub.com/ojiot/OJIOT_2020v6i1n07_Zeuch.html.

[294] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze. **Revisiting the Design of Data Stream Processing Systems on Multi-Core Processors**. In *ICDE*, pages 659–670. IEEE, 2017. doi:10.1109/ICDE.2017.119.

[295] S. Zhang, J. He, C. A. Zhou, and B. He. **BriskStream: Scaling Stream Processing on Multicore Architectures**. In *SIGMOD*, pages 705–722. ACM, 2019. doi:10.1145/3299869.3300067.

[296] S. Zhang, Y. Mao, J. He, P. M. Grulich, S. Zeuch, B. He, R. T. B. Ma, and V. Markl. **Parallelizing Intra-Window Join on Multicores: An Experimental Study**. In *SIGMOD*, pages 2089–2101. ACM, 2021. doi:10.1145/3448016.3452793.

[297] W. Zhang, J. Kim, K. A. Ross, E. Sedlar, and L. Stadler. **Adaptive code generation for data-intensive analytics**. *PVLDB*, 2021. doi:10.14778/3447689.3447697.

[298] J. Zhou, J. Cieslewicz, K. A. Ross, and M. Shah. **Improving database performance on simultaneous multithreading processors**. In *PVLDB*, pages 49–60, 2005. http://www.vldb.org/archives/website/2005/program/paper/tue/p49-zhou.pdf.

[299] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. **Dynamic plan migration for continuous queries over data streams**. In *SIGMOD*, pages 431–442. ACM, 2004. doi:10.1145/1007568.1007617.

[300] T. Ziegler, J. Nelson-Slivon, V. Leis, and C. Binnig. **Design Guidelines for Correct, Efficient, and Scalable Synchronization Using One-Sided RDMA**. *Proceedings of the ACM on Management of Data*, 1(2), jun 2023. https://doi.org/10.1145/3589276, doi:10.1145/3589276.

[301] A. Ziehn, P. M. Grulich, S. Zeuch, and V. Markl. **Bridging the Gap: Complex Event Processing on Stream Processing Engines**. In *Under Submission*.