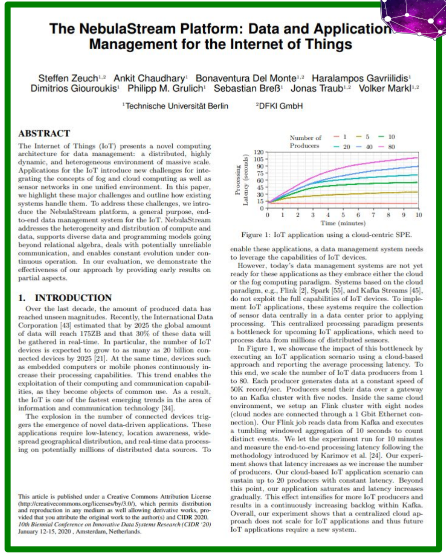
BIFOLD
The Berlin Institute for the Foundations of Learning and Data

Unifying Query Interpretation and Compilation

Philipp M. Grulich, Aljoscha Lepping, Dwi Nugroho, Varun Pandey,
Bonaventura Del Monte, Steffen Zeuch, Volker Markl

Three Query Compilers



The NebulaStream Platform: Data and Application Management for the Internet of Things

Steffen Zeuch[1,2] Ankit Chaudhary[1] Bonaventura Del Monte[1,2] Haralampos Gavriilidis[1] Dimitrios Giouroukis[1] Philipp M. Grulich[1] Sebastian Breß[1] Jonas Traub[1,2] Volker Markl[1,2]

[1]Technische Universität Berlin [2]DFKI GmbH

ABSTRACT

The Internet of Things (IoT) presents a novel computing architecture for data management: a distributed, highly dynamic, and heterogeneous environment of massive scale. Applications for the IoT introduce new challenges for integrating the concepts of fog and cloud computing as well as sensor networks in one unified environment. In this paper, we highlight these major challenges and outline how existing systems handle them. To address these challenges, we introduce the NebulaStream platform, a general purpose, end-to-end data management system for the IoT. NebulaStream addresses the heterogeneity and distribution of compute and data, supports diverse data and programming models going beyond relational algebra, deals with potentially unreliable communication, and enables constant evolution under continuous operation. In our evaluation, we demonstrate the effectiveness of our approach by providing early results on partial aspects.

1. INTRODUCTION

Over the last decade, the amount of produced data has reached unseen magnitudes. Recently, the International Data Corporation [43] estimated that by 2025 the global amount of data will reach 175ZB and that 30% of these data will be gathered in real-time. In particular, the number of IoT devices is expected to grow to as many as 20 billion connected devices by 2025 [21]. At the same time, devices such as embedded computers or mobile phones continuously increase their processing capabilities. This trend enables the exploitation of their computing and communication capabilities, as they become objects of common use. As a result, the IoT is one of the fastest emerging trends in the area of information and communication technology [34].

The explosion in the number of connected devices triggers the emergence of novel data-driven applications. These applications require low-latency, location awareness, widespread geographical distribution, and real-time data processing on potentially millions of distributed data sources. To

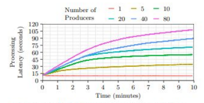


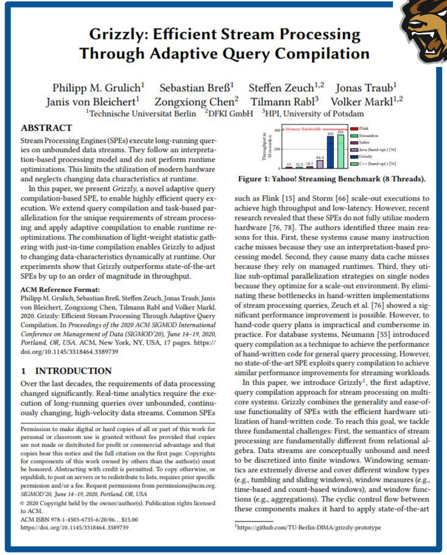
Figure 1: IoT application using a cloud-centric SPE.

enable these applications, a data management system needs to leverage the capabilities of IoT devices.

However, today's data management systems are not yet ready for these applications as they embrace either the cloud or the fog computing paradigm. Systems based on the cloud paradigm, e.g., Flink [2], Spark [55], and Kafka Streams [45], do not exploit the full capabilities of IoT devices. To implement IoT applications, these systems require the collection of sensor data centrally in a data center prior to applying processing. This centralized processing paradigm presents a bottleneck for upcoming IoT applications, which need to process data from millions of distributed sensors.

In Figure 1, we showcase the impact of this bottleneck by executing an IoT application scenario using a cloud-based approach and reporting the average processing latency. To this end, we scale the number of IoT data producers from 1 to 80. Each producer generates data at a constant speed of 50K record/sec. Producers send their data over a gateway to an Kafka cluster with five nodes. Inside the same cloud environment, we setup an Flink cluster with eight nodes (cloud nodes are connected through a 1 Gbit Ethernet connection). Our Flink job reads data from Kafka and executes a tumbling windowed aggregation of 10 seconds to count distinct events. We let the experiment run for 10 minutes and measure the end-to-end processing latency following the methodology introduced by Karimov et al. [24]. Our experiment shows that latency increases as we increase the number of producers. Our cloud-based IoT application scenario can sustain up to 20 producers with constant latency. Beyond this point, our application saturates and latency increases gradually. This effect intensifies for more IoT producers and results in a continuously increasing backlog within Kafka. Overall, our experiment shows that a centralized cloud approach does not scale for IoT applications and thus future IoT applications require a new system.

CIDR 2020



Grizzly: Efficient Stream Processing Through Adaptive Query Compilation

Philipp M. Grulich[1] Sebastian Breß[1] Steffen Zeuch[1,2] Jonas Traub[1] Janis von Bleichert[1] Zongxiong Chen[2] Tilmann Rabl[3] Volker Markl[1,2]
[1]Technische Universitat Berlin [2]DFKI GmbH [3]HPI, University of Potsdam

ABSTRACT

Stream Processing Engines (SPEs) execute long-running queries on unbounded data streams. They follow an interpretation-based processing model and do not perform runtime optimizations. This limits the utilization of modern hardware and neglects changing data characteristics at runtime.

In this paper, we present *Grizzly*, a novel adaptive query compilation-based SPE, to enable highly efficient query execution. We extend query compilation and task-based parallelization for the unique requirements of stream processing and apply adaptive compilation to enable runtime re-optimizations. The combination of light-weight statistic gathering with just-in-time compilation enables Grizzly to adjust to changing data-characteristics dynamically at runtime. Our experiments show that Grizzly outperforms state-of-the-art SPEs by up to an order of magnitude in throughput.

ACM Reference Format:
Philipp M. Grulich, Sebastian Breß, Steffen Zeuch, Jonas Traub, Janis von Bleichert, Zongxiong Chen, Tilmann Rabl and Volker Markl. 2020. Grizzly: Efficient Stream Processing Through Adaptive Query Compilation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3318464.3389739

1 INTRODUCTION

Over the last decades, the requirements of data processing changed significantly. Real-time analytics require the execution of long-running queries over unbounded, continuously changing, high-velocity data streams. Common SPEs

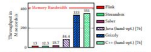


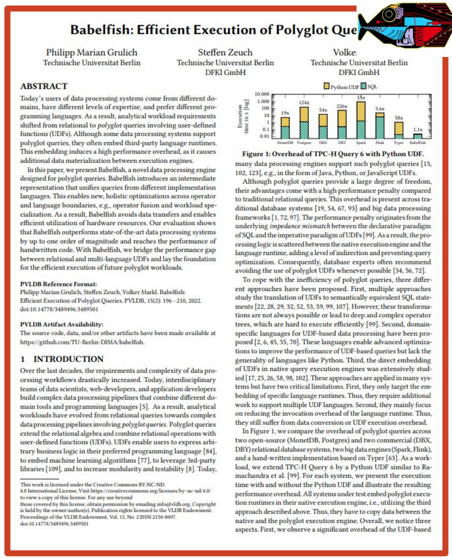
Figure 1: Yahoo! Streaming Benchmark (8 Threads).

such as Flink [15] and Storm [66] scale-out executions to achieve high throughput and low-latency. However, recent research revealed that these SPEs do not fully utilize modern hardware [76, 78]. 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. By eliminating these bottlenecks in hand-written implementations of stream processing queries, Zeuch et al. [76] showed a significant performance improvement is possible. However, to hand-code query plans is impractical and cumbersome in practice. For database systems, Neumann [55] introduced query compilation as a technique to achieve the performance of hand-written code for general query processing. However, no state-of-the-art SPE exploits query compilation to achieve similar performance improvements for streaming workloads.

In this paper, we introduce Grizzly[1], 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

[1]https://github.com/TU-Berlin-DIMA/grizzly-prototype

SIGMOD 2020



Babelfish: Efficient Execution of Polyglot Que

Philipp Marian Grulich, Technische Universitat Berlin
Steffen Zeuch, Technische Universitat Berlin, DFKI GmbH
Volke, Technische Universitat Berlin, DFKI GmbH

ABSTRACT

Today's users of data processing systems come from different domains, have different levels of expertise, and prefer different programming languages. As a result, analytical workload requirements shifted from relational to *polyglot* queries involving user-defined functions (UDFs). Although some data processing systems support polyglot queries, they often embed third-party language runtimes. This embedding induces a high performance overhead, as it causes additional data materialization between execution engines.

In this paper, we present Babelfish, a novel data processing engine designed for polyglot queries. Babelfish introduces an intermediate representation that unifies queries from different implementation languages. This enables new, holistic optimizations across operator and language boundaries, e.g., operator fusion and workload specialization. As a result, Babelfish avoids data transfers and enables efficient utilization of hardware resources. Our evaluation shows that Babelfish outperforms state-of-the-art data processing systems by up to one order of magnitude and reaches the performance of handwritten code. With Babelfish, we bridge the performance gap between relational and multi-language UDFs and lay the foundation for the efficient execution of future polyglot workloads.

PVLDB Reference Format:
Philipp Marian Grulich, Steffen Zeuch, Volker Markl. Babelfish: Efficient Execution of Polyglot Queries. PVLDB, 15(2): 196 - 210, 2022. doi:10.14778/3489496.3489501

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/TU-Berlin-DIMA/babelfish.

1 INTRODUCTION

Over the last decades, the requirements and 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 domain tools and programming languages [5]. As a result, analytical workloads have evolved from relational queries towards complex data processing pipelines involving *polyglot queries*. Polyglot queries extend the relational algebra and combine relational operations with user-defined functions (UDFs). UDFs enable users to express arbitrary business logic in their preferred programming language [84], to embed machine learning algorithms [77], to leverage 3rd-party libraries [109], and to increase modularity and testability [8]. Today,

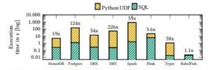


Figure 1: Overhead of TPC-H Query 6 with Python UDF.

many data processing engines support such polyglot queries [15, 102, 123], 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 [19, 54, 67, 93] and big data processing frameworks [1, 72, 97]. The performance penalty originates from the underlying *impedance mismatch* between the declarative paradigm of SQL and the imperative paradigm of UDFs [99]. As a result, the 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 often recommend avoiding the use of polyglot UDFs whenever possible [34, 56, 72].

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 [22, 28, 29, 32, 52, 53, 59, 99, 107]. However, these transformations are not always possible or lead to deep and complex operator trees, which are hard to execute efficiently [99]. Second, domain-specific languages for UDF-based data processing have been proposed [2, 6, 45, 55, 70]. 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 studied [17, 25, 26, 58, 98, 102]. 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 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 [63]. As a workload, we extend TPC-H Query 6 by a Python UDF similar to Ramachandra et al. [99]. 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

VLDB 2022

Take Away

Our query compilers enable **high performance** through **data and hardware-tailored specialization**.

Take Away

Our query compilers enable **high performance** through **data and hardware-tailored specialization.**

However, query compilers lead to **high system complexity** and require a **high engineering effort**!

Take Away

“[Query Compilation] is great for performance, but it is difficult for students to maintain and debug the code.”
April 2021, Database Deep Dives with Andy Pavlo

Decreasing Industry-Adoption

databricks

"Code generators are harder to build and debug then interpreted-engines."
Sigmod 2022, Photon: A Fast Query Engine for Lakehouse Systems

FIREBOLT

"Query Compilation increases engine complexity, makes it harder to onboard new engineers, and retain high development velocity."
PVLDB 2022, Photon: A Fast Query Engine for Lakehouse Systems

Velox

"Use cases where codegen provides clear benefits, outweighing compilation delays, decreased developer productivity, and debuggability are [still] under investigation."
PVLDB 2022, Velox: Meta's Unified Execution Engine

Let's take a step back!

Let's take a step back!

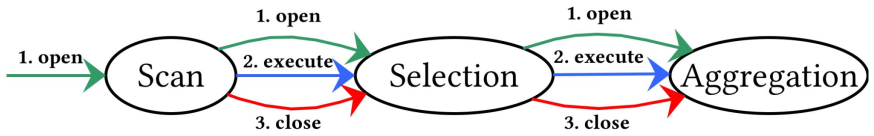
Could we unify interpretation and compilation?

Goals

Goals

1. Push-based query interpretation

- Alignes control and data-flow within execution.
- Fits well with task/morsel-driven parallelization.



Goals

1. Push-based query interpretation

2. Native Operator Implementations

- Support for standard control flow, virtual functions, abstractions.
- Native support for debugging and testing.

```
class Selection : public ExecutableOperator{
 void execute (RuntimeContext& ctx, Tuple& tuple){
  // calls child operator only if expression returns true
  if (expression->execute(tuple))
   child->execute(ctx, tuple);
}};

class LessThanExpression : public Expression{
 Value execute(Tuple& tuple){
  auto leftValue = leftSubExpression->execute(tuple);
  auto rightValue = rightSubExpression->execute(tuple);
  return leftValue < rightValue;
}};
```

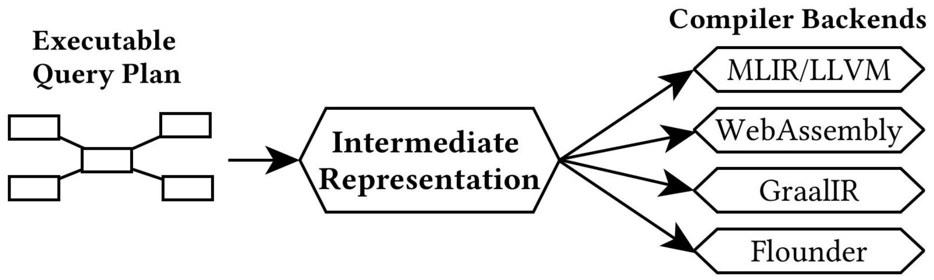
Goals

1. Push-based query interpretation

2. Native Operator Implementations

3. Automatic query compilation

- Generate IR from interpretation-based operators.
- Selects compilation backend depending on specific workload requirements.

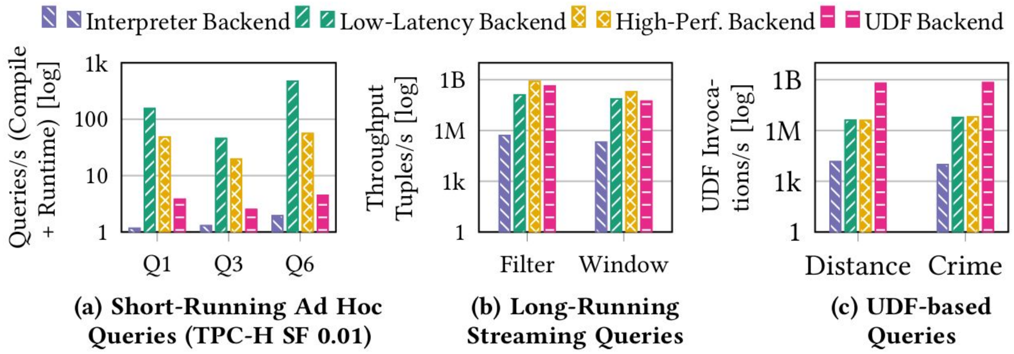


Conclusion

Summary:

✔ Framework with focus on developer experience.

✔ IR to target specialized code-generation backends.


Interpreter Backend
Low-Latency Backend
High-Perf. Backend
UDF Backend
Queries/s (Compile + Runtime) [log]
1k
Q1
Q3
Q6
(a) Short-Running Ad Hoc Queries (TPC-H SF 0.01)
Throughput Tuples/s [log]
1B
1M
1k
Filter
Window
(b) Long-Running Streaming Queries
UDF Invocations/s [log]
1B
1M
1k
Distance
Crime
(c) UDF-based Queries