Babelfish: Efficient Execution of Polyglot Queries

Philipp Marian Grulich
Technische Universitat Berlin

Steffen Zeuch
Technische Universitat Berlin
DFKI GmbH

Volker Markl
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,

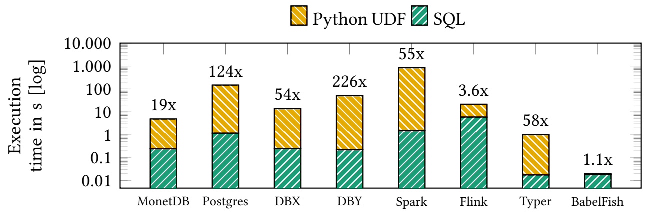


Figure 1: Overhead of TPC-H Query 6 with Python UDF.

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

Proceedings of the VLDB Endowment, Vol. 15, No. 2 ISSN 2150-8097.
doi:10.14778/3489496.3489501

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 paper, 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. 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 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.
(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 paper is structured as follows. First, we introduce a formal representation of polyglot queries in Section 2. Then, in Sections 3, we introduce a concept for the efficient execution of polyglot queries. Based on this concept, we describe Babelfish in detail in Section 4. In Section 5, we present optimizations to execute polyglot queries efficiently. Finally, we present our experiments in Section 6, related work in Section 7, and conclude in Section 8.

2 POLYGLOT QUERIES

Polyglot queries extend the relational algebra with UDFs and allow users to express processing logic in their preferred programming language. Listing 1 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 2.1 introduces a formal description of the structure of polyglot queries and individual operators. Based on this, Section 2.2 studies the data exchange between polyglot operators.

[1]Babelfish: the oddest thing in the Universe. Effectively removes all barriers to communications between different cultures and races [3].

```
-- 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 1: Polyglot Example Query.

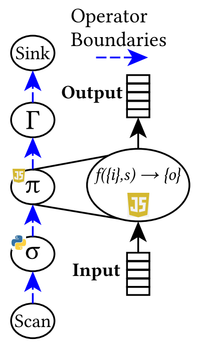


Figure 2: Query Tree.

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. [82], 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 $v \in V \setminus \{v_0\}$ 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 2 illustrates the query tree of Listing 1 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 2, embeds the `tripProfit()` UDF. During query processing, polyglot operators 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 [30] 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.

2.2 Modeling Data Exchange

Data exchange among polyglot operators defined in different programming languages requires data conversion. In the example query from Listing 1, 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 [43, 102]. 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.

3 THE CASE FOR POLYGLOT QUERIES

Many data processing systems support polyglot queries, e.g., MonetDB [98], Postgres [95], Exasol [75], Impala [118], or Flink [15]. 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 1. In the following, we first identify and analyze the main bottlenecks of polyglot query execution in current systems, see Section 3.1. Based on our analysis, we formulate design principles for a polyglot execution engine that addresses these bottlenecks, see Section 3.2.

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 [74]. Each invocation causes a substantial performance overhead due to virtual function calls that decrease code locality [102]. Research proposed to reduce the number of runtime invocations by batching the invocation of polyglot operators [98, 102]. However, calling the runtime is still required and decreases code efficiency. Furthermore, it introduces overhead for data exchange and data conversion.

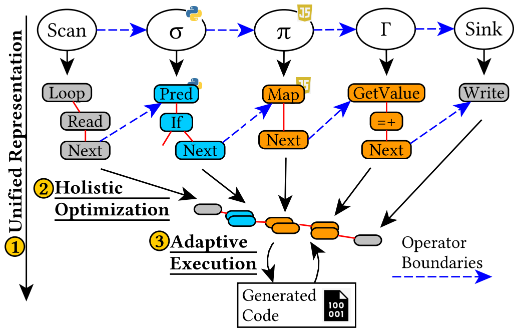


Figure 3: Polyglot query representation of Listing 1.

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.

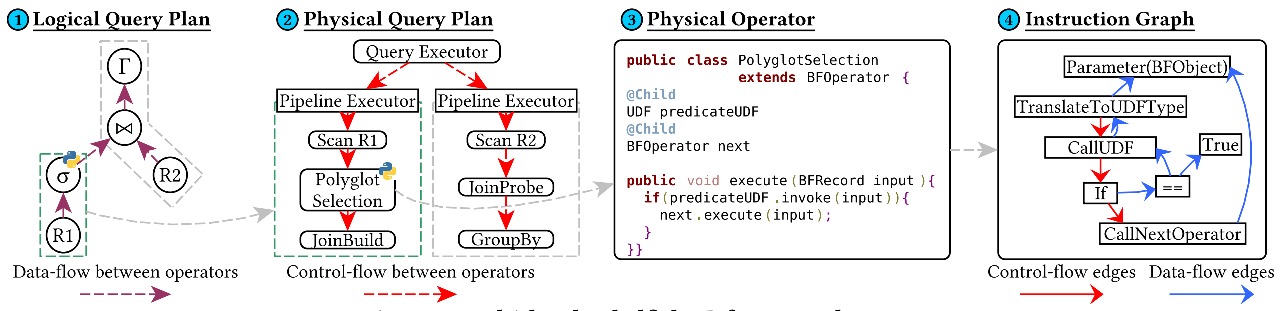
Bottleneck 3: Data Conversion. Polyglot operators define custom data types, see Section 2. This requires the execution engine to convert intermediate records from the CDR to the NDR of a particular operator before invoking it. Thus, the execution engine has to access each field and convert it to the 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.

3.2 Efficient Polyglot Query Execution

Section 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 3 using the query from Listing 1. 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 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.



```
public class PolyglotSelection
             extends BFOperator {
@Child
UDF predicateUDF
@Child
BFOperator next

public void execute(BFRecord input){
  if(predicateUDF.invoke(input)){
    next.execute(input);
  }
}}
```

Figure 4: Multi-level Babelfish-IR for example query.

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 3.1. For example, the optimizer could eliminate all operator boundaries for the query in Figure 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 [10]. 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 BABELFISH

This section introduces Babelfish^2, our novel data processing engine for polyglot queries. Babelfish applies the design principles from Section 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 [80] 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 [121] and Graal [27] 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 4.1, we introduce the Babelfish-IR and discuss the transformation of polyglot queries from the logical query plan to the final machine code. In Section 4.2, we introduce `BFRecords` to unify the data exchange among operators.

4.1 Query Representation

In Section 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 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 4.1.1), the *Physical Query Plan* 2 (see Section 4.1.2), a set of *Physical Operators* 3 (see Section 4.1.3), and the *Instruction Graph* 4 (see Section 4.1.4).

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 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.

4.1.2 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 [121]. Truffle provides an implementation framework for programming languages, e.g., GraalJS [87] and Graal-Python [86]. Based on Truffle, Babelfish represents the PQP, provides implementations of relational operators, and handles the interaction between operators and UDFs.

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.

^{2}The source code is available at https://github.com/TU-Berlin-DIMA/babelfish.

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 [48], 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, 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 [87] and Python [86] 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.

4.1.4 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 [27], which is a graph-based IR for generic programs in static single assignment (SSA) form [24]. 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, 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 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.

4.2 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 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 4.2.1). Based on this, we present `PolyRecords` for the data exchange among built-in operators and UDFs (see Section 4.2.2). Finally, we describe the mapping of `BFRecords` to a physical memory layout (see Section 4.2.3).

4.2.1 BFRecords. Babelfish introduces `BFRecords` to represent data exchange between operators. `BFRecords` are record types [16] that define a collection of fields and each field consists of a name and a type, i.e., $r_i : \langle name_i : type_i \rangle$. 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.

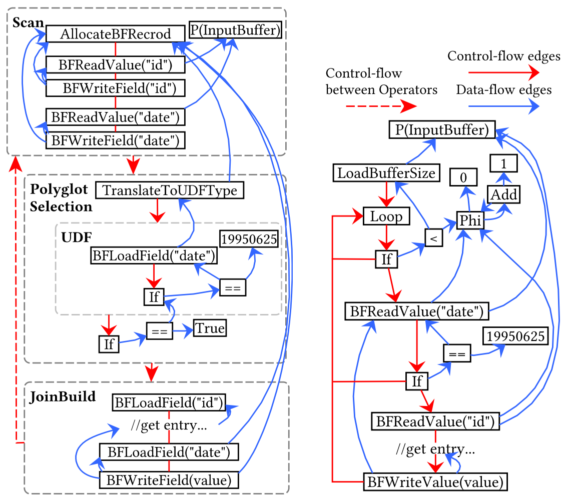
Primitive Types. Babelfish supports common primitive data types, e.g., Boolean, Int, or Double. These types build the foundation for composed data types and are crucial for primitive expressions, e.g., `linenumber > 2`. Operations on these types usually directly correspond to hardware operations in the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

4.2.2 PolyRecords. 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 [49] 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



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

Figure 5: Instruction graph for build pipeline of Figure 4.

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.

4.2.3 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.

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 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 5.2. Second, we introduce *latch-reduction* to improve the efficiency of stateful polyglot operators in concurrent environments in Section 5.3. Third, we present *workload specializations* to improve the efficiency of raw and textual data processing in Section 5.4.

5.1 Eliminating Operator Boundaries

In Section 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 [80] 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 [13, 50, 81, 82, 92].

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 5, we illustrate operator fusion for the build pipeline from Figure 4. The pipeline consists of three physical operators, a `Scan`, a `PolyglotSelection`, and a `JoinBuild`. In Figure 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 [122]. Partial evaluation combines the code of a generic program with runtime constant input data to create a specialized program [42, 60]. 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 [120].

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

[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" [37]

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 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 [73], 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* [89, 122]. 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 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 [80].

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

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 [126]. To mitigate this overhead in relational data processing engines, research proposed branch-free operator implementations, using *predication* [14, 103, 128]. 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.

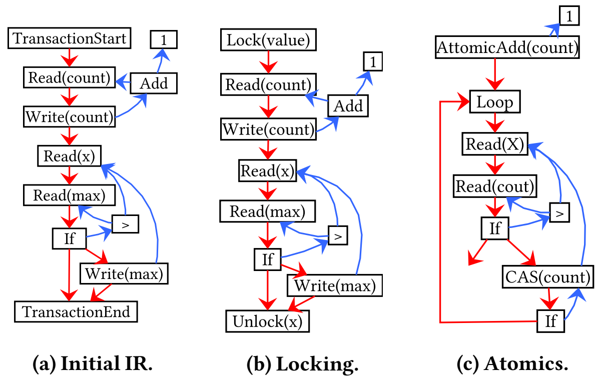


Figure 6: Operator-IR of stateful UDF.

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.

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 [7, 15]. 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 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 6b. With high contention, latches may cause a significant overhead [12]. 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 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 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 6.3.3).

5.4 Workload Specialization

Modern data analytic workloads drastically differ from traditional relational queries. They often contain text-heavy computations [117] or directly process raw data [62]. 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 5.4.1 and the support of raw data in Section 5.4.2.

5.4.1 Text Processing. UDFs often analyze or manipulate text values, e.g., word-count, tokenization, or n-gram computation [70]. 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 [9]. 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 6.3.4).

5.4.2 Raw Data Processing. Modern data processing workloads often directly process raw files in third-party data formats, e.g., CSV or Arrow [39]. Before processing, most systems convert raw data to an internal data format. This requires data parsing and materialization, which induces a high overhead [57]. To mitigate this, Babelfish proposes *lazy parsing* and interleaves parsing with query processing.

Lazy Parsing. In Section 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 6.3.5).

6 EVALUATION

In this section, we experimentally evaluate different aspects of Babelfish. We introduce our experimental setup in Section 6.1. After that, we conduct two sets of experiments. In Section 6.2, we compare the performance of Babelfish across multiple workloads to state-of-the-art data processing systems. In Section 6.3, we perform micro-experiments to study specific aspects of Babelfish.

6.1 Experimental Setup

Throughout our evaluation, we use the hardware and software configurations that are described in Section 6.1.1 and run the workloads on the datasets that are described in Section 6.1.2.

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.

6.1.2 Datasets. 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* [83], which was used in multiple publications to assess the performance of big data systems on common data science workloads [70, 96, 116]. 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.

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 6.2.1), data science workloads (see Section 6.2.2), and UDFs that embed 3rd-party libraries (see Section 6.2.3).

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 [63]. Similar to Ramachandra et al. [99], we replace operators with either Python, JavaScript, or Java UDFs within Babelfish. As baselines, we evaluate all queries without UDFs on MonetDB [11], a general-purpose DBMS, Pandas [76], a

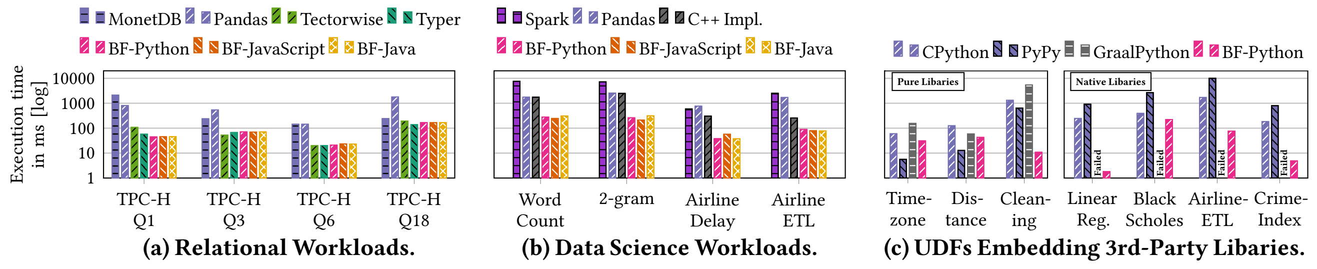


Figure 7: Comparison of Babelfish with hand optimized baselines across workloads.

common data processing framework, and Typer/Tectorwise [63] as hand-optimized C++ implementations of the above queries.

Results. Figure 7a shows execution times of MonetDB, Pandas, Typer, Tectorwise, and Babelfish for the selected queries. Across all queries, Babelfish outperforms MonetDB/Pandas 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. [63]. 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. [63]. 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.

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. [96], we implement each query as a sequence of Python, JavaScript, and Java UDFs. As a baseline, we evaluate the selected queries on Spark [124] and Pandas [76], as common data science frameworks, and a hand-written C++ implementation.

Results. In Figure 7b 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 [33, 127]. 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-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.

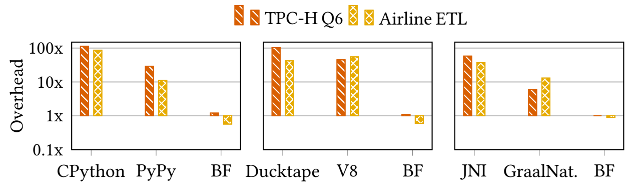
Summary. This experiment showed that Babelfish reaches high performance on data science workloads. Babelfish benefits from the optimization of polyglot queries, which fuse operators, eliminate allocations, and specialize operations within UDFs. As a result, Babelfish outperforms data science frameworks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

6.2.3 *Embedding 3rd-party libraries in UDFs.*

A common use case of UDFs is the embedding of 3rd-party libraries [109]. Therefore, we evaluate seven queries [90, 96] 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 [36] (the standard Python runtime), PyPy [10, 112] (a high performance Python JIT-Compiler), and GraalPython (Babelfish's underlying Python runtime).

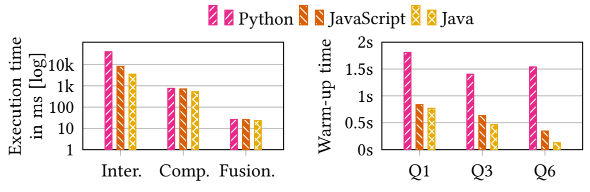
Results. Figure 7c shows that Babelfish outperforms CPython across all queries, whereby the speedups depends on the actual workload. Queries that embed pure Python libraries benefit from JIT compilation, i.e., PyPy outperforms the other Python runtimes by up to 5x. Due to the prototypical state of GraalPython, 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.

For queries that embed native Python extensions, JIT compilers have to (re)implement the C-API of CPython [38]. 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 [108] and is a focus of the HPy project [113]. To mitigate this problem, Babelfish



(a) Python UDF. (b) JavaScript UDF. (c) Java UDF.

Figure 8: Overhead of language runtimes and Babelfish.



(a) Execution Mode. (b) Warm-up time.

Figure 9: Impact of adaptive query compilation.

leverages the BFRecords to substitute calls into these libraries with build-in Babelfish functions. This approach is similar to Weld [90] and enables Babelfish to fuse operations, 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.

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 6.3.1. Based on this, we study the effect of Babelfish's JIT compilation on execution performance and warm-up time in Section 6.3.2. Then, we evaluate the handling of stateful UDFs in multi-core environments in Section 6.3.3. Finally, we analyze Babelfish's performance for specific workloads, i.e., string operations in Section 6.3.4 and raw data processing in Section 6.3.5.

6.3.1 Language Runtimes. In our initial experiment in Figure 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 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 6.2 with state-of-the-art language runtimes. In particular, we use CPython [36] and PyPy [10, 112] for Python UDFs, V8 [47] and Duktape [115] for JavaScript UDFs, and HotSpot JNI [89] and GraalNative [85] for Java UDFs.

Results. In Figure 8, 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.

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). 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 9a, 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 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 9b, 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 [110] and DBLAB [106]. 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 [110]. Consequently, Babelfish is applicable for a wide range of UDF-based use-cases.

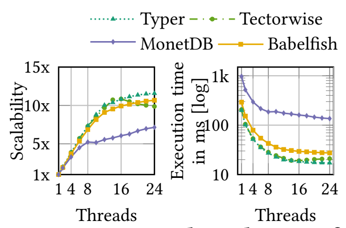


Figure 10: Scaling degree of parallelism on TPC-H Q6.

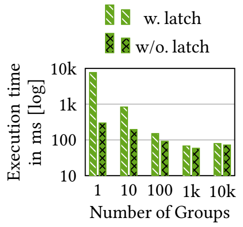


Figure 11: Contention handling.

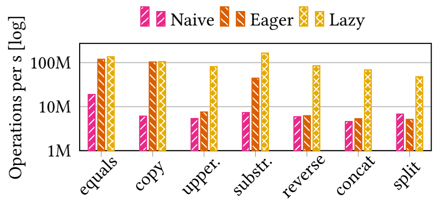


Figure 12: Text processing.

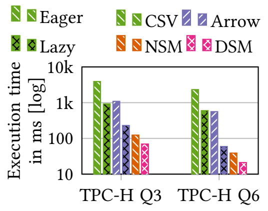


Figure 13: Raw data processing.

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 Babelfish's *latch-reduction* optimization for stateful UDFs, as proposed in Section 5.4.1.

Results. In Figure 10, we evaluate the scalability across all systems with 1 to 24 execution threads. Overall, Babelfish achieves a similar speedup as Typer and Tectorwise (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 11, 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.

6.3.4 *Text Processing.* Text manipulation causes a high performance overhead in current data processing systems, as shown in Figure 7b. To mitigate this bottleneck, Babelfish introduces `PolyglotRopes` to perform text processing lazily (see Section 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 12, 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 6.2. As a result, Babelfish is applicable for a wide range of data science tasks, which are usually very text-heavy.

6.3.5 *Raw Data Processing.* Polyglot workloads often process raw data [57], e.g., CSV or Arrow files. To enable high performance across different data formats, Babelfish interleaves data parsing and processing (see Section 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 buffer before processing.

Results. In Figure 13, we make three observations. First, processing CSV data causes the highest execution time, as Babelfish scans the whole file to infer record boundaries and 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.

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, e.g., text manipulations, Babelfish even outperforms hand-optimized implementations as Babelfish is able to perform optimizations across operator boundaries.

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. [100], Krikellas et al. [68], and Neumann [80]. Over the last decade, it was applied in many data processing systems [4, 40, 68, 78, 80, 81, 91, 92, 119] and to different workloads, e.g., stream [50, 114, 125] and spatial data processing [111]. 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 [65] and DBLAB [106], Babelfish defines the Babelfish-IR as a central representation of queries. Similar to LB2 [110], 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. Another line of research focused on reducing the time required for query compilation [41, 64, 66]. Similar to Kohn et al. [66], Babelfish applies JIT compilation. Overall, Babelfish leverages query compilation as a building block to enable the holistic 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 [101]. 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 [20–22, 28, 29, 31, 32, 99, 107] or particular 3rd-party libraries [53, 59, 61] to equivalent relational expressions. With the integration of Froid [99] in Microsoft's SQL Server [88], this approach received a lot of attention. Froid converts loop-free UDFs into plain SQL queries. Based on this, Gupta et al. [52] proposed Aggify to optimize loops in UDFs. Duta et al. [28, 29] 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 [99]. 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 [2, 6, 44–46, 51, 55, 69]. They utilize an IR to enable advanced query optimizations, e.g., loop fusion and dead code elimination [2], selection pushdown [51], optimizations for distributed dataflows [6, 44–46], or the integration of different algebras [69]. 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 [18, 25, 35, 43, 70, 75, 79, 98]. 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 [75], which causes a high overhead for calling the external runtime. More advanced approaches leverage strided execution models to reduce this overhead [98, 102]. In contrast, Babelfish embeds polyglot operators directly in the physical execution plan to eliminate these boundaries. Ishizaki et al. [58], Trill [17, 18], and Gerenuk [79] 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. [105] and Tupleware [23] 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 [94]. Second, the translation of UDF languages to LLVM-IR is a challenging problem by itself. Even a mature LLVM-based compiler like Numba [71] 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 [70], specializing CSV parsing [104], applying predication on JavaScript programs [128]. 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 efficent execution.

8 CONCLUSION

In this paper, 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 polyglot queries. As a result, Babelfish enables the efficient execution of polyglot queries independent of the definition language of individual 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 future polyglot workloads. As future work, we plan to incorporate Babelfish in our new data processing platform NebulaStream [125].

ACKNOWLEDGMENTS

This work was funded by the DFG Priority Program (MA4662-5), the German Federal Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A) and Software Campus (01IS17052). Furthermore, we thank Gábor E. Gévay and the Graal community.

REFERENCES

[1] 2016. Spark functions vs UDF performance? https://stackoverflow.com/questions/38296609/spark-functions-vs-udf-performance.

[2] Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. 2012. Jet: An embedded DSL for high performance big data processing. In *BigData*.

[3] Douglas Adams. 1979. *The Hitchhiker's Guide to the Galaxy*. Pan Books.

[4] Sameer Agarwal, Davies Liu, and Reynold Xin. 2016. 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. [Online; accessed 31.5.2019].

[5] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, et al. 2018. RHEEM: enabling cross-platform data processing: may the big data be with you! *PVLDB* (2018).

[6] Alexander Alexandrov, Andreas Kunft, Asterios Katsifodimos, Felix Schüler, Lauritz Thamsen, Odej Kao, Tobias Herb, and Volker Markl. 2015. Implicit parallelism through deep language embedding. In *SIGMOD*. ACM.

[7] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *SIGMOD*. ACM.

[8] Suresh Basinasetty. 2021. User Defined Functions in SQL. https://www.tutorialgateway.org/user-defined-functions-in-sql/.

[9] Hans-J Boehm, Russ Atkinson, and Michael Plass. 1995. Ropes: an alternative to strings. *Software: Practice and Experience* (1995).

[10] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS*.

[11] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*.

[12] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *DaMoN*. ACM.

[13] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating custom code for efficient query execution on heterogeneous processors. *VLDBJ* (2018).

[14] David Broneske, Sebastian Breß, and Gunter Saake. 2013. Database scan variants on modern CPUs: A performance study. In *In Memory Data Management and Analysis*. Springer.

[15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *TCDE* (2015).

[16] Luca Cardelli and John C Mitchell. 1991. Operations on records. *Mathematical structures in computer science* (1991).

[17] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, Danyel Fisher, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. *PVLDB* (2014).

[18] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, and James F Terwilliger. 2015. Trill: Engineering a Library for Diverse Analytics. *IEEE Data Engineering Bulletin* (2015).

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

[20] Alvin Cheung, Owen Arden, Samuel Madden, Armando Solar-Lezama, and Andrew C Myers. 2013. StatusQuo: Making Familiar Abstractions Perform Using Program Analysis.. In *CIDR*.

[21] Alvin Cheung, Samuel Madden, Armando Solar-Lezama, Owen Arden, and Andrew C Myers. 2014. Using Program Analysis to Improve Database Applications. *Data Engineering Bulletin* (2014).

[22] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *SIGPLAN Notices*. ACM.

[23] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. 2015. An Architecture for Compiling UDF-centric Workflows. *PVLDB* (2015).

[24] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS* (1991).

[25] Mohammad Dashti, Sachin Basil John, Thierry Coppey, Amir Shaikhha, Vojin Jovanovic, and Christoph Koch. 2018. Compiling Database Application Programs. *CIDR* (2018).

[26] Bin Dong, Kesheng Wu, Surendra Byna, Jialin Liu, Weijie Zhao, and Florin Rusu. 2017. ArrayUDF: User-Defined Scientific Data Analysis on Arrays. In *HPDC*. ACM.

[27] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. [n.d.]. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *VMIL*. ACM.

[28] Christian Duta and Torsten Grust. 2020. Functional-Style SQL UDFs With a Capital 'F'. In *SIGMOD*. ACM.

[29] Christian Duta, Denis Hirn, and Torsten Grust. 2019. Compiling PL/SQL Away, In CIDR. *arXiv preprint arXiv:1909.03291*.

[30] Andrew Eisenberg and Jim Melton. 1999. SQL: 1999, formerly known as SQL3. *ACM Sigmod record* (1999).

[31] K. Venkatesh Emani, Tejas Deshpande, Karthik Ramachandra, and S. Sudarshan. 2017. DBridge: Translating Imperative Code to SQL. In *SIGMOD*. ACM.

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

[33] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In *OSDI*. USENIX.

[34] Claudio Davide Ferrara. [n.d.]. Light up the Spark in catalyst by avoiding UDF. https://www.codemotion.com/magazine/light-up-the-spark-in-catalyst-by-avoiding-udf-4061.

[35] Yuan Yu Michael Isard Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, and Pradeep Kumar Gunda Jon Currey. 2009. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *LSDS-IR* (2009).

[36] Python Software Foundation. 2020. CPython. https://github.com/python/cpython.

[37] Python Software Foundation. 2020. Duck-Typing. https://docs.python.org/3/glossary.html#term-duck-typing.

[38] Python Software Foundation. 2021. C-API. https://docs.python.org/3/c-api/.

[39] The Apache Software Foundation. 2021. Apache Arrow. https://arrow.apache.org/.

[40] Craig Freedman, Erik Ismert, and Per-Åke Larson. 2014. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Engineering Bulletin* (2014).

[41] Henning Funke, Jan Mühlig, and Jens Teubner. 2020. Efficient Generation of Machine Code for Query Compilers. In *DaMoN*. ACM.

[42] Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, computers, controls* (1971).

[43] Haralampos Gavriilidis. 2019. Computation Offloading in JVM-based Dataflow Engines. *BTW* (2019).

[44] Gábor E. Gévay, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. The Power of Nested Parallelism in Big Data Processing – Hitting Three Flies with One Slap –. In *SIGMOD*. ACM.

[45] Gábor E Gévay, Tilmann Rabl, Sebastian Breß, Loránd Madai-Tahy, and Volker Markl. 2018. Labyrinth: Compiling imperative control flow to parallel dataflows. *arXiv preprint arXiv:1809.06845* (2018).

[46] Gábor E Gévay, Tilmann Rabl, Sebastian Breß, Loránd Madai-Tahy, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Efficient control flow in dataflow systems: When ease-of-use meets high performance. In *ICDE*. IEEE.

[47] Google. 2020. v8. https://v8.dev/.

[48] Goetz Graefe. 1994. Volcano/spl minus/an extensible and parallel query evaluation system. *TKDE* (1994).

[49] Matthias Grimmer, Chris Seaton, Roland Schatz, Thomas Würthinger, and Hanspeter Mössenböck. 2015. High-performance cross-language interoperability in a multi-language runtime. In *Symposium on Dynamic Languages*.

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

[51] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. 2012. Spotting code optimizations in data-parallel pipelines through periscope. In *OSDI*.

[52] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. 2020. Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates. In *SIGMOD*.

[53] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. 2021. Putting Pandas in a Box.. In *CIDR*.

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

[55] Arvid Heise, Astrid Rheinländer, Marcus Leich, Ulf Leser, and Felix Naumann. 2012. Meteor/sopremo: An extensible query language and operator model. In *BigData*. Citeseer.

[56] IBM. 2020. 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.

[57] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. 2011. Here are my data files. here are my queries. where are my results?. In *CIDR*.

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

[59] Alekh Jindal, K Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Mueller, et al. 2021. Magpie: Python at speed and scale using cloud backends. In *CIDR*.

[60] Neil D. Jones. 1996. An Introduction to Partial Evaluation. *Comput. Surveys* (1996).

[61] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan,

Markus Weimer, et al. 2020. Extending relational query processing with ML inference. In *CIDR*.
[62] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. 2016. Fast Queries over Heterogeneous Data through Engine Customization. *PVLDB* (2016).
[63] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *PVLDB* (2018).
[64] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra. *VLDB J.* (2021).
[65] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. In *PVLDB*. VLDB Endowment.
[66] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive execution of compiled queries. In *ICDE*. IEEE.
[67] Hugo Kornelis. 2012. T-SQL User-Defined Functions: the good, the bad, and the ugly. https://sqlserverfast.com/blog/hugo/2012/05/t-sql-user-defined-functions-the-good-the-bad-and-the-ugly-part-1/.
[68] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *ICDE*. IEEE.
[69] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An intermediate representation for optimizing machine learning pipelines. *PVLDB* (2019).
[70] Andreas Kunft, Lukas Stadler, Daniele Bonetta, Cosmin Basca, Jens Meiners, Sebastian Breß, Tilmann Rabl, Juan José Fumero, and Volker Markl. 2018. ScootR: Scaling R Dataframes on Dataflow Systems.. In *SoCC*. ACM.
[71] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A llvm-based python jit compiler. In *Workshop on the LLVM Compiler Infrastructure in HPC*. 1–6.
[72] Jacek Laskowski. 2021. 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.
[73] David Leopoldseder, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 2018. A Cost Model for a Graph-based Intermediate-representation in a Dynamic Compiler. In *VMIL*.
[74] Sheng Liang. 1999. *The Java native interface: programmer's guide and specification*. Addison-Wesley Professional.
[75] Stefan Mandl, Oleksandr Kozachuk, and Jens Graupmann. 2017. Bring Your Language to Your Data with EXASOL. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)* (2017).
[76] Wes McKinney et al. 2010. Data structures for statistical computing in python. In *SCIPY*.
[77] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research* (2016).
[78] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, and Andrew Pavlo. 2020. Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling. *PVLDB* (2020).
[79] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. 2019. Gerenuk: Thin Computation over Big Native Data Using Speculative Program Transformation. In *SOSP*. ACM.
[80] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *PVLDB* (2011).
[81] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*.
[82] Thomas Neumann and Guido Moerkotte. 2009. Generating optimal DAG-structured query evaluation plans. *Computer Science-Research and Development* (2009).
[83] Bureau of Transportation Statistics. 2020. Reporting Carrier On-Time Performance. https://www.transtats.bts.gov/Tables.asp?DB_ID=120. [Online; accessed 22.2.2021].
[84] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. 2008. Pig latin: a not-so-foreign language for data processing. In *SIGMOD*. ACM.
[85] Oracle. 2020. Graal Native Image. https://www.graalvm.org/reference-manual/native-image/.
[86] Oracle. 2020. Graal Python. https://github.com/graalvm/graalpython.
[87] Oracle. 2020. GraalJS. https://github.com/graalvm/graaljs.
[88] Brent Ozar. 2019. Froid: How SQL Server 2019 Will Fix the Scalar Functions Problem. https://www.brentozar.com/archive/2018/01/froid-sql-server-vnext-might-fix-scalar-functions-problem/.
[89] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The java hotspot TM server compiler. In *JVM*. USENIX.
[90] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimajan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman Amarasinghe, et al. 2018. Evaluating end-to-end optimization for data analytics applications in weld. *PVLDB* 11, 9 (2018), 1002–1015.
[91] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. 2017. Weld: A common runtime for high performance data analytics. In *CIDR*.
[92] Paroski Paroski. 2016. Code generation: The inner sanctum of database performance. http://highscalability.com/blog/2016/9/7/code-generation-the-inner-sanctum-ofdatabase-performance.html. [Online; accessed 31.5.2019].
[93] Glenn Paulley. [n.d.]. http://glennpaulley.ca/database/2015/07/performance-overhead-of-sql-user-defined-functions/.
[94] Holger Pirk, Jana Giceva, and Peter R Pietzuch. 2019. Thriving in the No Man's Land between Compilers and Databases.. In *CIDR*.
[95] PostgreSQL. 2020. PostgreSQL. https://www.postgresql.org/.
[96] Vignesh Prajapati. 2013. *Big data analytics with R and Hadoop*. Packt Publishing Ltd.
[97] Farooq Qaiser. 2018. UDFs vs. Map vs. Custom Spark-Native Functions. https://medium.com/@Farox2q/udfs-vs-map-vs-custom-spark-native-functions-91ab2c154b44.
[98] Mark Raasveldt and Hannes Mühleisen. 2016. Vectorized UDFs in Column-Stores. In *SSDBM*. ACM.
[99] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: optimization of imperative programs in a relational database. *PVLDB*.
[100] Jun Rao, Hamid Pirahesh, C Mohan, and Guy Lohman. 2006. Compiled query execution engine using JVM. In *ICDE*. IEEE.
[101] Astrid Rheinländer, Ulf Leser, and Goetz Graefe. 2017. Optimization of Complex Dataflows with User-Defined Functions. *Comput. Surveys* (2017).
[102] Viktor Rosenfeld, Rene Mueller, Pinar Tözün, and Fatma Özcan. 2017. Processing Java UDFs in a C++ Environment. In *SoCC*. ACM.
[103] Kenneth A. Ross. 2002. Conjunctive Selection Conditions in Main Memory. In *SIGMOD*. ACM.
[104] Filippo Schiavio, Daniele Bonetta, and Walter Binder. 2020. Towards Dynamic SQL Compilation in Apache Spark. In *‹Programming›*. ACM, New York, NY, USA.
[105] Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. 2020. Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In *SSDBM*. ACM.
[106] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to architect a query compiler. In *SIGMOD*.
[107] Varun Simhadri, Karthik Ramachandra, Arun Chaitanya, Ravindra Guravannavar, and S Sudarshan. 2014. Decorrelation of user defined function invocations in queries. In *ICDE*. IEEE, 532–543.
[108] Victor Stinner. 2019. Python performance Past, Present, Future. In *EuroPython*.
[109] Jincheng Sun and Markos Sfikas. 2021. PyFlink: The integration of Pandas into PyFlink. https://flink.apache.org/2020/08/04/pyflink-pandas-udf-support-flink.html.
[110] Ruby Y Tahboub, Grégory M Essertel, and Tiark Rompf. 2018. How to architect a query compiler, revisited. In *SIGMOD*.
[111] Ruby Y. Tahboub and Tiark Rompf. 2020. Architecting a Query Compiler for Spatial Workloads. In *SIGMOD*. ACM.
[112] PyPy Team. 2020. PyPy. https://foss.heptapod.net/pypy/pypy.
[113] The HPy Team. 2021. HPy: a better API for Python. https://github.com/hpyproject/hpy.
[114] Georgios Theodorakis, Alexandros Koliousis, Peter Pietzuch, and Holger Pirk. 2020. LightSaber: Efficient Window Aggregation on Multi-Core Processors. In *SIGMOD*. ACM.
[115] Sami Vaarala. 2020. Duktape. https://duktape.org/.
[116] Shivaram Venkataraman, Zongheng Yang, Davies Liu, Eric Liang, Hossein Falaki, Xiangrui Meng, Reynold Xin, Ali Ghodsi, Michael Franklin, Ion Stoica, et al. 2016. Sparkr: Scaling r programs with spark. In *SIGMOD*. ACM.
[117] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. 2018. Get real: How benchmarks fail to represent the real world. In *DBTest*.
[118] Skye Wanderman-Milne and Nong Li. 2014. Runtime Code Generation in Cloudera Impala. *IEEE Data Engineering Bulletin* (2014).
[119] Skye Wanderman-Milne and Nong Li. 2014. Runtime Code Generation in Cloudera Impala. *IEEE Data Engineering Bulletin* (2014).
[120] Henry S Warren. 2013. *Hacker's delight*. Pearson Education.
[121] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-Optimizing Runtime System. In *SPLASH*. ACM.
[122] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical partial evaluation for high-performance dynamic language runtimes. In *ACM SIGPLAN*.
[123] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. USENIX.
[124] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* (2010).
[125] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform: Data and application

management for the internet of things. In *CIDR*.
[126] Steffen Zeuch and Johann-Christoph Freytag. 2015. Selection on modern cpus. In *IMDM*. ACM.
[127] Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. Analyzing efficient stream processing on modern hardware. *PVLDB* (2019).
[128] Wangda Zhang, Junyoung Kim, Kenneth A Ross, Eric Sedlar, and Lukas Stadler. 2021. Adaptive code generation for data-intensive analytics. *VLDB* (2021).