Query Compilation Without Regrets

PHILIPP M. GRULICH, Technische Universität Berlin, Germany
ALJOSCHA P. LEPPING, Technische Universität Berlin, Germany
DWI P. A. NUGROHO, Technische Universität Berlin, Germany
BONAVENTURA DEL MONTE*, Observe Inc., USA
VARUN PANDEY, Technische Universität Berlin, Germany
STEFFEN ZEUCH, Technische Universität Berlin, Germany
VOLKER MARKL, Technische Universität Berlin, Germany and DFKI GmbH, Germany

Engineering high-performance query execution engines is a challenging task. Query compilation provides excellent performance, but at the same time introduces significant system complexity, as it makes the engine hard to build, debug, and maintain. To overcome this complexity, we propose Nautilus, a framework that combines the ease of use of query interpretation and the performance of query compilation. On the one hand, Nautilus provides an interpretation-based operator interface that enables engineers to implement operators using imperative C++ code to ensure a familiar developer experience. On the other hand, Nautilus mitigates the performance drawbacks of interpretation by introducing a novel trace-based, multi-backend JIT compiler that translates operators into efficient code. As a result, Nautilus bridges the gap between compilation and interpretation and provides the best of both worlds, achieving high performance without sacrificing the productivity of engineers.

CCS Concepts: • **Information systems → DBMS engine architectures**.

Additional Key Words and Phrases: query compilation, query execution, database engines

ACM Reference Format:
Philipp M. Grulich, Aljoscha P. Lepping, Dwi P. A. Nugroho, Bonaventura Del Monte, Varun Pandey, Steffen Zeuch, and Volker Markl. 2024. Query Compilation Without Regrets. *Proc. ACM Manag. Data* 2, 3 (SIGMOD), Article 165 (June 2024), 28 pages. https://doi.org/10.1145/3654968

1 INTRODUCTION

Cloud vendors, like Snowflake [19], Amazon [4], or Databricks [6], build high-performance query execution engines to elastically scale a variety of data processing workloads. The main engineering challenge for these engines is to balance performance and productivity of the developers. On the one hand, an engine has to provide high-performance query execution for a wide range of workloads from various end users [6]. To achieve this, system engineers have to develop efficient data processing operators, which involve traditional relational operators as well as specialized operators for stream

*Work was conducted while the author worked at the Technische Universität Berlin.

Authors' addresses: Philipp M. Grulich, grulich@tu-berlin.de, Technische Universität Berlin, Berlin, Germany; Aljoscha P. Lepping, aljoscha.p.lepping@tu-berlin.de, Technische Universität Berlin, Berlin, Germany; Dwi P. A. Nugroho, d.nugroho@tu-berlin.de, Technische Universität Berlin, Berlin, Germany; Bonaventura Del Monte, ventura@observeinc.com, Observe Inc., USA; Varun Pandey, varun.pandey@tu-berlin.de, Technische Universität Berlin, Berlin, Germany; Steffen Zeuch, steffen.zeuch@tu-berlin.de, Technische Universität Berlin, Berlin, Germany; Volker Markl, volker.markl@tu-berlin.de, Technische Universität Berlin, Berlin, Germany and DFKI GmbH, Berlin, Germany.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2836-6573/2024/6-ART165
https://doi.org/10.1145/3654968

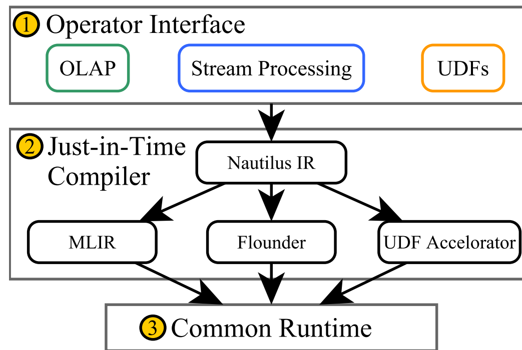


Fig. 1. Overview of the Nautilus framework.

processing, or user-defined functions (UDFs) [30]. On the other hand, the engine must be maintainable by a large number of engineers to ensure high productivity for the timely integration of new features. To this end, the engine has to be easy to modify, test, and debug [6]. Consequently, engine builders need to compromise between performance and maintainability, depending on their specific requirements.

Over the last decade, *vectorized query interpretation* [9] and *query compilation* [55] have emerged as the state-of-the-art architectures for high-performance query execution engines. Vectorized query interpretation extends the traditional Volcano processing model [27] and passes vectors of records between precompiled operators that can be developed in imperative code. Therefore, engineers can extend and debug the engine in their well-known programming workflow using standard developed tools and debuggers. In contrast, *query compilation* translates queries into code at runtime. This enables the data processing engine to generate specialized machine code that reaches high execution performance at the cost of an additional compilation-time overhead. To compile data processing queries, research has proposed different specialized query compilation strategies. These approaches either optimize for short-running queries [24, 39], target specific hardware [65], propose performance optimizations [18, 50], or focus on particular workloads [16, 29, 30]. Even though query compilers enable high performance, their development and maintenance is complex, and thus data processing systems struggle with their integration [6].

Engineers of state-of-the-art query compilers face two key challenges: First, query compilers generate the implementation of operators only after query submission, i.e., at runtime. This introduces an indirection between the implementation and the execution of operators, which makes compilation-based engines hard to build and debug [6, 63]. This is particularly problematic for query compilers that use specialized compiler frameworks like LLVM as they require engineers to have a deep understanding of compiler technology [41]. Second, the design space of a query compiler is very large, and there is no approach that is optimal for all workloads [28]. Thus, engineers must build workload-specific query compilers that balance compilation time, execution performance, and engineering effort [39]. From a company perspective, it is difficult to hire and onboard qualified engineers that fullfil this highly specialized profile [53, 63]. These challenges motivated recent commercial data management systems like Photon [6] and FireBolt [63] to adopt the vectorized query interpretation in contrast to query compilation to ensure quick prototyping, high engineer productivity, and consequently rapid development.

To address these challenges, we propose a novel way for systems to benefit from the advantages of query compilation without sacrificing developer productivity. In particular, we propose Nautilus[1], a

[1]We provide Nautilus as a stand-alone query compilation framework and use it as a foundation for our open-source data processing system NebulaStream. Access the code at github.com/nebulastream/nautilus.

framework for developing data processing engines that bridges the gap between query interpretation and query compilation. To this end, Nautilus combines an interpretation-based programming model with a novel trace-based, just-in-time (JIT) compiler that provides multiple backends to translate operators into efficient code. Figure 1 shows the three main components of Nautilus. First, (1) Nautilus provides a generic interface for implementing diverse data processing operators. The interface hides the complexity of code generation and enables engineers to write imperative C++ code that is easy to develop, debug, and maintain. Second, (2) Nautilus' JIT compiler traces imperative operators to derive a unified intermediate representation, the Nautilus IR, and provides multiple execution backends to produce efficient code. This enables Nautilus to specialize query execution towards particular workload requirements, e.g., minimizing startup latency or maximizing execution performance. Third, (3) Nautilus introduces a common interface between the executable operators and the host runtime that can be used across all execution backends. This simplifies the implementation of operators and enables engineers to reuse common data structures, e.g., hash-tables, lists, across operators. Our evaluation shows that Nautilus reduces the complexity of compilation-based execution engines and achieves high performance for relational-, streaming-, and UDF-based workloads. As a result, Nautilus combines the ease of use and productivity of query interpretation with the flexibility and excellent performance of state-of-the-art query compilers, enabling query compilation without regrets.
In summary, our contributions are as follows:

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

The rest of this paper is structured as follows: First, we discuss the challenges of query compilation in execution engines (see Section 2). Based on this, we introduce Nautilus (see Section 3), our operator implementation interface (see Section 4), and our novel trace-based, multi-backend JIT-compiler (see Section 5). Then, we evaluate Nautilus across different workloads (see Section 6). Finally, we discuss related work (see Section 8), and conclude (see Section 9).

2 THE CURSE OF QUERY COMPILATION

Over the last decade, many data processing systems applied query compilation to maximize query execution performance [3, 4, 51, 55]. Even though query compilation is widely adopted, it introduces a high system complexity and decreases engineering productivity. The engineers at Databricks recently discussed the engineering challenges of their query compiler for SparkSQL [6]. They argue that with a vectorized, interpretation-based architecture, it is easier to develop and scale the engine. Similarly, several recently introduced commercial systems, such as Photon [6], Firebolt [63], and Velox [64] followed interpretation-based architectures to reduce development costs and ensure the productivity of their engineers. In particular, the following two challenges hinder the adoption of query compilation.

Challenges 1: Managing engineering complexity. Building and maintaining query execution engines, like any other software artifact, requires developer time and costs. A survey on the software industry reveals that 50% of the overall expenditure (1.25 trillion US dollars at the time) went towards developing and debugging software [12]. Consequently, it is necessary to be frugal and minize developement costs, as well as maximize developer productivity. Operators in interpretation-based engines correspond to straightforward code fragments that are developers can comprehend and debug without additional tools. Listing 1 illustrates the C++ code of a vectorized aggregation. In contrast, query compilers use code generation frameworks such as LLVM [55] or build custom

Listing 1. C++ Example.

```
1 int64_t exec(int64_t* ptr, int64_t size){
2   int64_t sum = 0;
3   for (int64_t i = 0; i < size; i++) {
4     sum = sum + ptr[size];
5   }
6   return sum;
7 }
```

Listing 2. LLVM Example.

```
1 define i64 @exec(i64 %0, ptr %1) {
2   br label %3
3   3:; preds = %9, %2
4   %4 = phi i64 [ %17, %9 ], [ 0, %2 ]
5   %5 = phi i64 [ %18, %9 ], [ 0, %2 ]
6   %6 = phi i64 [ %10, %9 ], [ %0, %2 ]
7   %7 = phi ptr [ %13, %9 ], [ %1, %2 ]
8   %8 = icmp slt i64 %5, %6
9   br i1 %8, label %9, label %19
10  9:; preds = %3
11  %10 = phi i64 [ %6, %3 ]
12  %11 = phi i64 [ %5, %3 ]
13  %12 = phi i64 [ %4, %3 ]
14  %13 = phi ptr [ %7, %3 ]
15  %14 = mul i64 %11, 8
16  %15 = getelementptr i8,
17        78 ptr %13, i64 %14
18  %16 = load i64, ptr %15, align 4
19  %17 = add i64 %12, %16
20  %18 = add i64 %11, 1
21  br label %3
22  19:; preds = %3
23  %20 = phi i64 [ %4, %3 ]
24  ret i64 %20
25 }
```

Table 1. Compilation Backends.

	Throughput	Latency	Compl.
Programming Languages			
JAVA-BC [1]	♦	▼	▲
C/C++ [7, 29, 31, 80]	▲▲	▼▼	▲
OpenCL/ CUDA [11, 25, 65]	▲▲	▼▼	▲
Compiler Frameworks			
MLIR [36, 37]	▲▲	▲	▼
LLVM [50, 54, 57]	▲▲	▲	▼▼
WASM [32]	▲	▲▲	▼▼
ASM-JIT [24]	♦	▲▲	▼▼

compilers [24, 39] to generate code at runtime. Such code often resembles assembly and is highly complex, see generated LLVM-IR in Listing 2. Although both representations are semantically equivalent, understanding and debugging the generated code is cumbersome for most engineers [80]. Additionally, operating with existing tools (such as debuggers, stack trace tools etc.) at runtime is challenging without manually adding instrumentation. All these additional overheads, makes query compilation based engine hard to build, debug, and maintain. As a result, it becomes hard to find engineers that have the required expertise [63] and increases the development costs as reported by engineers at Databricks [6]. This is particularly problematic for academic projects like Mutable [32], NoisePage [51], or Peleton [50], that can't find contributors, as many students struggle with the complexity of query compilation [53]. To overcome this challenge, a compilation-based engine has to provide a framework that focuses on productivity and hides code generation complexity.

Challenges 2: Navigating a large design space. Modern data processing systems support an increasingly diverse set of workloads and hardware. To address different workload requirements, research introduced specialized query compilers, e. g., for short-running queries [39], stream processing [29, 75], user-defined functions [16, 30], and heterogeneous hardware [65]. These compilers generate code in different programming languages or use specialized compiler frameworks, to trade-off between compilation time, execution performance, and developer productivity as illustrated in Table 1. However, no architecture is suitable for all workloads. As a result, system engineers must develop and maintain different query compilation backends to efficiently support diverse workloads. For instance, Umbra provides multiple compilation backends to support short- and long-running queries [39]. To navigate this design space a compilation-based engine should offer a framework that easily allows to integrate different execution backends that optimize for specific workloads.

Both challenges increase the engineering effort and the cost of compilation-based query execution engines. Besides commercial vendors, this also impacts academia as most research groups cannot afford such engineering efforts [53]. As a result, the consensus has emerged that (i) compilation-based engines can reach superior execution performance, but (ii) interpretation-based engines are much easier to build and maintain. Consequently, getting the best of both worlds remains a desirable goal. To this end, we propose Nautilus, our query compilation framework that bridges the gap between query interpretation and compilation.

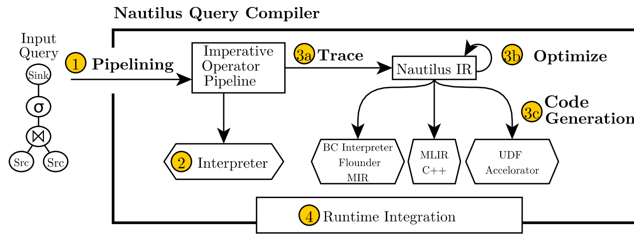


Fig. 2. Overview of query compilation Nautilus.

3 QUERY COMPILATION WITH NAUTILUS

In this section, we present Nautilus, our novel and extensible framework for implementing compilation-based query execution engines. Nautilus addresses the previously outlined challenges and enables engineers to utilize query compilation without compromising productivity. In particular, Nautilus decouples the implementation of operators from the actual query execution by adhering to the following three aspects: *1.* Nautilus provides an imperative interface to implement data processing operators that ensure developer productivity. *2.* Nautilus provides a novel trace-based JIT compiler that tailors query execution towards specific workload requirements. *3.* Nautilus allows the seamless integration with host data processing systems. This versatility enables engineers to employ Nautilus either as a foundation for new execution engines or as a specialized accelerator for distinct workloads in existing engines.

The remainder of this section provides an overview of Nautilus' architecture (see Section 3.1) and its extensibility (see Section 3.2). After that, we discuss Nautilus' operator implementation interface (see Section 4) and its JIT compiler in detail (see Section 5).

3.1 Architecture of Nautilus

In general, we assume that Nautilus is embedded into a host system either as part of the query execution engine or as specialized accelerators. The host system provides an already optimized query plan that can contain relational-, streaming-, or UDF-based operators. This allows Nautilus to support queries for a wide range of workloads **(C2)**. Depending on the specific workload and environment, Nautilus' select an execution strategy and creates an executable query plan (illustrated in Figure 2).

In the first step (1), Nautilus translates the query plan into pipelines [55] of executable Nautilus operators that provide the operator implementation. Next, Nautilus selects either an interpretation (2) or compilation (3) - based execution strategy to trade-off between debuggability and performance. Finally (4), Nautilus creates the executable query plan, which tightly integrates with the runtime of the host system. In the following, we discuss these aspects in detail.

3.1.1 Operator Implementation. Developing operators for compilation-based data processing engines is complex as discussed in Section 2. In particular, the indirection of generating operator implementations at runtime makes development challenging (**C1**) and hinders the support for diverse workloads (**C2**). To facilitate ease of development, Nautilus provides an intuitive interface for operator implementations that targets three design goals: First, it follows an interpretation-based processing model that decouples individual operators from each other. Second, it allows engineers to implement operators in generic imperative C++ code using lightweight abstractions that are easy to test and debug. Third, it provides common data types and allows a tight integration with the host system. Using this interface, engineers can implement operators without reasoning about code generation, while it enables Nautilus to automatically generate efficient code at runtime.

3.1.2 Interpretation-based Query Execution. A major challenge of compilation-based execution engines is the limited support for debugging the implementation of operators during development [40]. To tackle this challenge, Nautilus provides a dedicated interpretation-based execution strategy that executes operators directly without involving any code generation. This strategy initializes pipelines of operators and applies a traditional tuple-at-a-time model. At runtime, system engineers can use debuggers like gdb to follow the execution path of records and inspect values as well as operator state. In our experience, this is crucial to investigate logical errors in operator implementations and improves developer productivity significantly.

3.1.3 Compilation-based Query Execution. In contrast to traditional query compilers, Nautilus' executable operators and data structures are implemented in generic and imperative C++ code. To achieve high performance, Nautilus introduces a novel trace-based JIT compilation approach, which translates operators to efficient code in three steps. Initially, Nautilus symbolically executes operator pipelines and collects all executed operations in a trace object. Then, it converts the trace to a Nautilus IR fragment. This IR allows for further optimizations as well as separates the implementation of operators from the final code generation. In the final step, Nautilus selects one of multiple compilation backends to generate executable code. These backends are tailored to specific workload characteristics, i.e., Nautilus provides low-latency backends to target short-running queries, high-performance backends for long-running queries, and specialized backends to accelerated specific workloads like UDFs.

3.1.4 Runtime Integration. The integration between operators and the host system is a crucial factor that affects the maintainability of compilation-based execution engines. For the general case, Nautilus exposes pre-defined runtime functions to operators and transparently converts intermediate values at the function boundary. While this method ensures a clear separation of concerns and increases testability, it also introduces function calls in the generated code that may induce a performance cost. To mitigate this overhead, Nautilus provides two strategies. First, Nautilus' JIT compiler can automatically inline runtime function during code generation at the cost of an increasing compilation time. Second, Nautilus' operator implementation interface also supports the implementation of complex algorithms and data structures. While this requires additional engineering effort, it enables the compiler to produce one unified code fragment that avoids any function calls to the runtime. This often enables further compilation optimizations and increases code efficiency. As a result, Nautilus allows engineers to balance performance and engineering complexity and to focus on performance-critical areas.

3.2 Extensibility

The target workloads of data processing systems may broaden over the course of their lifetime to cover new use cases. For example, Velox [64] was extended with support for ML-workloads and Vector datatypes. As a result, system engineers have to adjust and extend the functionality of the data processing engine (**C2**). To this end, Nautilus provides a flexible *plugin* interface that can be extended. Plugins enables system engineers to extend Nautilus with new functionality on three levels. First, plugins can provide new strategies for the instantiation of executable operators and query segmentation, for instance, to use hybrid pipelining strategies [16, 18, 50]. Second, plugins can define new operators, expressions, or data types. To this end, Nautilus provides handlers to intercept logical and arithmetical operations during execution. This enables a plugin for specialized data types, such as spatial data types, to reuse expressions and operators. Third, plugins can extend Nautilus' JIT compiler and provide new compiler backends to accelerate specific workloads, like UDF acceleration using specialized compilers.

In summary, plugins increase the extensibility Nautilus-based execution engines. They enable the reuse of components, improve the testability of the resulting engine, and increase productivity.

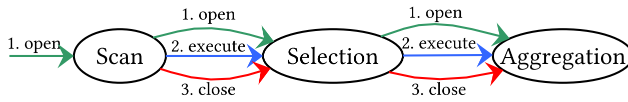


Fig. 3. Execution of operators within a pipeline.

4 OPERATOR IMPLEMENTATION INTERFACE

Nautilus provides an intuitive operator implementation interface that relieves engineers from the complexity of traditional compilation-based data processing engines. It follows an interpretation-based processing model and allows engineers to implement operators in generic imperative C++ code using lightweight abstractions that are easy to test and debug. Using this interface, we implemented common relational data processing operators, e.g., projections, joins, aggregations, as well as logical and arithmetical expressions. In this section, we present individual aspects of this interface and use the operator implementation of TPC-H Q6 as a running example. This query first scans the input data (see Listing 3), performs a set of selections (see Listing 4), and finally aggregates an expression (see Listing 6).

4.1 Pipeline Evaluation

Nautilus segments query plans into a set of pipelines. During query execution, each pipeline receives data from predecessor pipelines and emits intermediate results to successor pipelines. Within pipelines, Nautilus follows a push-based execution model [55] and sends data from one operator to another via function calls (illustrated in Figure 3). In contrast to the pull model of traditional interpretation-based engines [27], the push model aligns the control- and data-flow between operators. Both follow the same direction, i.e., from the initial `Scan` towards the most downstream operator. This simplifies the implementation and debugging of individual operators (**C1**).

For the implementation of operators, Nautilus defines an intuitive interface, similar to the Volcano model [27]. Operators may implement `setup()`, `teardown()`, `open()`, and `close()` to specify their processing logic. Each method encapsulates a specific step in the execution lifecycle of an operator and helps to keep the implementation maintainable. `Setup()` and `teardown()` are called once per operator and initialize/clear global operator state, for example the hash-table in a grouped aggregation. `Open()` and `close()` are invoked for each buffer of data and allow operators to maintain local states, for instance, an emit operator allocates an output buffer to materialize results. Furthermore, the `open()` function of a scan operator iterates over the input buffer. It extracts individual tuples that they pass to its children, see Listing 3. These children implement the `execute()` function to process individual tuples. For example, the selection in Listing 4 evaluates an expression on each input tuple. Additionally, operators receive a reference to the `RuntimeContext`. This context maintains the operator state and provides access to the runtime system to allocate data structures and coordinate processing across threads. As a result, Nautilus' combination of a push-based processing model and an intuitive operator interface decouples operators, simplifies their implementation, and improves testability (**C1**).

4.2 Imperative Operator Implementation

For the implementation of individual operators, Nautilus provides a lightweight C++ interface that focuses on simplicity and expressiveness (**C1**). Operators can be entirely implemented in generic, imperative C++ code. Thus, engineers are relieved from the complexity of code generation and can express data processing logic using common data types, function calls, and control-flow statements, e.g., **`if`**, **`for`**, or **`while`**. For example, the `Scan` operator uses a simple **`for`** loop that iterates over the content of a data buffer and uses virtual function calls to operate on the input buffer (see Line 7 in

Listing 3. Scan.

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

Listing 4. Selection.

```
1 class Selection : public ExecutableOp{
2  void execute (RuntimeCtx& ctx, Tuple& t){
3   // calls child operator if
4   // expression is true
5   if (expression->execute(t))
6    child->execute(ctx, t);
7 }};
8
9 class LessThan : public Expression{
10  Value execute(Tuple& t){
11   auto leftValue = leftExp->execute(t);
12   auto rightValue = rightExp->execute(t);
13   return leftValue < rightValue;
14 }};
```

Listing 5. Aggregation.

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

Listing 6. Sum Function.

```
1 class Sum : public Aggregation{
2  Value lift(Tuple& t){
3   return inputExp->execute(t);
4  }
5  void update(State* state, Value& v){
6   state->sum += v;
7  }
8  Value lower(State* state){
9   return state->sum;
10 }};
```

Listing 3). To express data processing operations, Nautilus provides three core abstractions, i. e., `TupleBuffers`, `Tuples`, and `Values`.

`TupleBuffers` represent chunks of memory that store data according to a specific column or row-oriented data layout and provide methods to read and write tuples at particular positions.

`Tuples` represent individual data entries as record types and define a collection of field names and associated values. Operators receive data as tuples and read/write values by their field names.

`Values` represent data elements of a particular type and can be part of a `Tuple` or the result of an operation, e.g., the evaluation of the `LessThanExpression` in Listing 4. As `Values` contain a concrete data element, system engineers can directly inspect their content at runtime in a debugger like gdb. `Values` can either be primitive types, e.g., `Int8`, `Double`, `Ptr`, collection types, e.g., `Array<Int64>`, or composed types, e.g., `Point`. All Primitive value types directly map to an associated C++ type, `Value<Int8>`→**`int8_t`** and behave semantically the same. Nautilus uses operator overloading to provide logical and arithmetical operations between `Values`. Thus, engineers can use `Values` similarly to standard C++ data types. For example, the `LessThanExpression` in Listing 4 Line 13 evaluates < on its inputs and returns the result as a `Value`.

These abstractions decouple the implementation of operators from the physical data representation, which has multiple benefits that improve the maintainability of an engine: First, Nautilus' operators only receive `Tuples`, which are independent of the data layout of a `TupleBuffer`. Thus, engineers can add different physical data layouts without adjusting the implementation of all operators. Second, `Values` can be used to implement complex data types that combine multiple `Values` and provide specialized functions. For instance, Nautilus' `Text` type maintains the text length as well as a data pointer internally and provides common text manipulation functions. Third, these abstractions enable engineers to split complex operators into individual components, i.e., sub-operators [44]. One

Listing 7. Hash Join Probe.

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

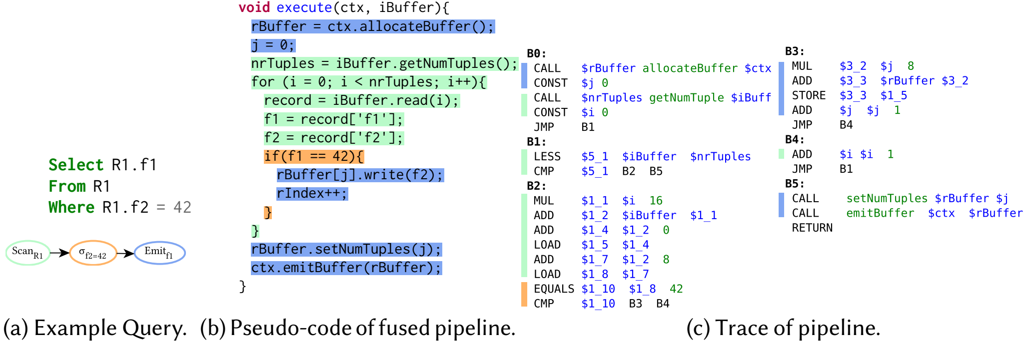
Listing 8. Map Interface.

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

example is the Aggregation operator in Listing 5. It maintains a global state in the `RuntimeContext` and passes each tuple to a set of aggregation functions in Line 6. All aggregation functions implement the same generic interface as proposed by Tangwongsan et al. [74], and provide three functions `lift()`, `combine()` and `lower()` see Listing 6. Lift transforms a tuple to a partial aggregate. Combine computes the combined aggregate from partial aggregates and updates the current aggregation state. Lower transforms a partial aggregate to a final aggregate. This allows for the reuse of aggregation function implementation across various physical operators, such as keyed and global aggregation for batch data and window aggregations in stream processing.

4.3 Integrating Data Structures

In addition to the implementation of individual processing logic, the integration between operators and complex data structures is crucial for the architecture of an engine. To this end, Nautilus provides common data structures, like `Lists` and `HashTables`, which can be used across different physical operators. At the core, all these data structures are implemented by a runtime component on the one side and a Nautilus interface on the other side. The runtime component is pre-compiled, while the interface creates function calls to invoke specific functions on the data structure. This design allows engineers to balance performance and engineering complexity. For performance-critical code, they can implement parts of the data structure in Nautilus, which enables the compiler to generate specialized code at runtime. For all other functions, they can directly call into the runtime code and benefit from the increased debuggability and testability of pre-compiled code. To illustrate this, let us consider the `HashJoinProbe` operator in Listing 7. For each input tuple, it accesses entries in the `Hashmap` from Listing 8 to identify join partners. First, the operator selects key values (Lines 4-7) and calculates the hash (Line 9). Subsequently, it invokes `findOne(hash, keyValues)` on the hash map data structure. Listing 8 shows the implementation of this function using a simple chained hash map. Initially, it carries out a function call in the runtime to locate the chain corresponding to the specific hash value (`findChain()`). Then, it iterates through all entries in the chain, comparing their keys. If



(a) Example Query. (b) Pseudo-code of fused pipeline. (c) Trace of pipeline.

Fig. 4. Illustration of the intermediate trace (c) for an example query (a). The query performs a scan, selection, and an emit operator, which execute the set of operations on Value objects illustrated in the pseudo-code of the fused pipeline (b). The resulting IR is shown in Figure 5.

a matching entry is found, it returns to the HashJoinProbe operator. During compilation, Nautilus generates a single code fragment for both the HashJoinProbe operator and the findOne function. Only the implementation of findChain() is pre-compiled and linked.

Since the Hashmap is defined through a generic interface, its concrete implementation is decoupled from individual operators.

5 TRACE-BASED JUST-IN-TIME COMPILATION

Nautilus operators are implemented in generic and imperative C++ code to ensure high developer experience and provide no code-generation logic. To translate operators to efficient code, Nautilus introduces trace-based JIT compilation for data processing queries.

The main idea of a tracing JIT compiler is to dynamically optimize hot code paths during the execution of a program [5]. To this end, it first executes a program in an interpreter and records all executed instructions (the so-called *trace*). If the same trace was executed multiple times, e.g., because it traces a loop, the compiler translates the trace to machine code. In this case, the generated code only covers the hot code path of the original program. Today, this technique is the foundation of many mature JIT compilers like PyPy [8] for Python and TraceMonkey [26] for JavaScript.

In contrast to these general-purpose compilers, Nautilus operates on operator pipelines, which always contain a tight loop over some data, i.e., a scan and a set of operators that process individual records. As the shape of pipelines and the set of operators is restricted, we can eliminate the need for the initial interpretation to detect hot-code paths. Instead, Nautilus uses symbolic execution [10] to trace all possible execution paths through an operator pipeline. This enables to translate operator pipelines to efficient code in three steps: *1)* Nautilus uses a linear algorithm to record a trace of all the instructions an operator pipeline executes. *2)* Nautilus translates the trace to its intermediate representation, i.e., the Nautilus IR. *3)* Nautilus generates efficient machine code using specialized compilation backends. This enables Nautilus to generate efficient code from imperative operators and to balance compilation time and execution performance.

In the following, we discuss tracing, our Nautilus IR, and the compilation backends in detail. Figure 4 illustrates a running example of the individual steps of Nautilus' trace-based JIT compiler. From the initial query (a) Nautilus creates an executable query plan that fuses individual operators to data-centric pipelines (b). During JIT compilation, Nautilus executes the pipeline symbolically, creates the trace (c), and converts it to a Nautilus IR fragment (see Figure 5).

Algorithm 1 Trace Pipeline

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

Algorithm 2 Trace Operation

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

5.1 Tracing Data-Processing Queries

In order to enable the tracing of data processing queries, Nautilus follows three key observations: *1)* Operators within a query are constant and do not change during execution. *2)* The targets of function calls between operators and the host runtime are constant. *3)* Operations that do not depend on any input data, represented by `Value` objects, can not change during query execution and are constant. This allows us to differentiate between operations that are *runtime constant* and *runtime dynamic* during tracing. If an operation involves `Value` objects it becomes *runtime dynamic*. In any other case we assume it to be *runtime constant*. For example, the `scan` in Figure 4 accesses a constant number of fields (`f1` and `f2`) for each tuple, whereas the actual values of the individual attributes are only determined at runtime, i.e., they are *runtime dynamic*. This distinction enables Nautilus to efficiently trace the implementation of operators. During tracing, Nautilus follows Algorithm 1 and 2 and executes pipelines *symbolically* to record all runtime dynamic operations that involve `Value` objects in a lightweight trace object.

Symbolic tracing. To derive the trace of an operator pipeline, Nautilus's tracing algorithm executes pipelines multiple times using dummy data[2]. In each execution, it intercepts all operations that involve `Value` objects and records them in the trace, e.g., the expression `f1 == 42` in Line 9 results in an `EQUALS` instruction in the trace. These instructions capture references to input and result `Values` as well as unique `tags` to identify if the same operation was executed multiple times, e.g., as part of a loop in the `scan` operator. As the trace captures only the operations and their dependencies, the actual data values do not impact the trace and are not recorded.

A critical requirement for tracing is to capture all potential execution paths during query evaluation. For instance, in our running example, the trace has to contain both outcomes and the resulting control flow of the `if` statement in the `selection` operator. To this end, Algorithm 1 evaluates the pipeline till all execution paths have been visited. It maintains a set of operation `tags` Θ and a queue of in-flight execution paths E. Each execution path $\epsilon \in E$ captures a unique sequence of operations that have been executed during query evaluation. Till all execution paths have been visited ($E \neq \emptyset$), Nautilus continuous to evaluate the pipeline. For each traced operation, Algorithm 2 checks if the operation is part of a loop or if it causes a control-flow split, e.g., by a `if` statements. Control-flow splits require another pipeline evaluation such that Nautilus can also trace the other control-flow branch. To this end, Nautilus checks if the operation was executed before ($op_{tag} \in \Theta$). If it is executed for the first time, it appends the current execution path ϵ to the set of in flight executions E and continues evaluating the `true` case. If it visits the split for the second time, it evaluates the `false` case in contrast. If it revisits the same control-flow split, Nautilus terminates the pipeline evaluation as both paths are already part of

[2]Tracing only collects the operations that are executed on dynamic data values. Thus, Nautilus executed the pipeline using "dummy" null values during symbolic tracings.

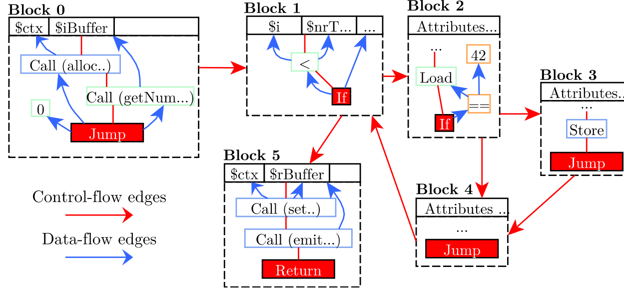


Fig. 5. Illustration of the Nautilus IR for the example query and trace from Figure 4.

the trace. As a result, the tracing algorithm requires $O(2n)$ iterations, respectively pipeline evaluations, in the worst case to evaluate all execution paths for a pipeline with n control-flow splits. Thus, even a complex pipeline with multiple nested operators requires only a small number of iterations.

In our running example, Nautilus traces the operation of the `scan`, `select`, `emit` operators as part of the same pipeline. As the function calls between operators are constant, tracing automatically fuses the operators, which results in the code illustrated in Figure 4 b. For each executed operation, Nautilus adds an instruction to the trace (see Figure 4 c). Both the `scan` and `select` operator introduce a control-flow split. In the first iteration, Nautilus enters the loop body of the `scan`, evaluates the `true` case of the `selection`, and exits the loop as it executes the loop header (`i < nrTuples`) for the second time. Consequently, Nautilus only requires a second iteration to evaluate the `false` case of the `selection` and can terminate tracing early. The resulting trace covers all visited execution paths through the query and represents control flow via branches and basic blocks.

5.2 Nautilus IR

For each trace Nautilus generates a Nautilus IR fragment. The Nautilus IR is our unified intermediate representation to decouple the implementation of operators from a specific compilation backend. Nautilus IR focuses on three aspects to simplify the implementation of compilation backends: *1)* It is agnostic to specific operators and compilation backends. *2)* It focuses on a small set of build-in operations and data types to simplify the backend code. *3)* It supports transformations to enable optimizations independent of a backend. Thus, Nautilus IR balances compactness and expressiveness.

Nautilus IR follows static single-assignment (SSA) form and differentiates between functions, basic blocks, and operations (see IR in Figure 5). Functions usually correspond to operator pipelines and contain a sequence of basic blocks representing the control flow (illustrated with red arrows). Each basic block receives block arguments, defines a sequence of dataflow operations, and terminates with a control-flow operation. Operations may depend on input operations and produce at most one result value of a primitive type (illustrated with blue arrows). To this end, Nautilus IR provides common data flow operations, i.e., logical and arithmetical expressions, function calls, load, and stores, as well as control-flow operations for jumps and if conditions. To pass values between basic blocks, each block can define a set of block arguments [48]. In contrast to traditional SSA ϕ nodes, this models dependencies between values and operations explicitly and simplifies optimizations.

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

Table 2. Comparison of Nautilus' compilation backends.
(▲ indicates good, ♦ indicates medium, ▼ indicates low)

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

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

5.3 Compilation Backends

Previous query compilation approaches proposed different intermediate representations and compilation backends to optimize for specific workloads, e.g., short-running batch queries, long-running stream processing queries, or the acceleration of UDFs (**C2**). To this end, they made specific trade-offs between the throughput of the generated code, compilation latency, and ease of use.

In contrast, Nautilus leverages its backend-independent IR to specialize query execution towards the requirements of specific workloads at runtime. To implement a compilation backend Nautilus provides a generic interface for engineers. Each backend receives Nautilus IR fragments of operator pipelines and returns executable code. The simplicity of the Nautilus IR facilitates the implementation of individual backends. As the backends only translate Nautilus IR instructions to one specific code-gerneration target they consist of relatively few code (couple of hundreds lines) with low complexity.

Nautilus provides three types of backends with different performance characteristics and code complexity, illustrated in Table 2: *low-latency backends*, which minimize compilation time for short-running workloads; *high-performance backends*, which maximize throughput for long-running workloads; and *specialized backends*, which accelerate specific workloads, e.g., the execution of UDFs.

Each Nautilus backend represents a specific spot in the design space of a query compiler and has unique performance characteristics, i.e., favoring throughput or latency. This flexibility highlights the versatility of Nautilus' compilation approach and enables the comparison of established query compilation approaches from literature and to propose new ones. Currently, Nautilus uses simple heuristics to choose between the backends, e.g., data set size or the presence of UDFs. In the future, we plan to leverage runtime adaptivity in Nautilus to select the optimal execution strategy [43]. In the remainder of this section, we discuss the individual backends in detail.

5.3.1 Low-Latency Backends. The compilation latency of a query compiler significantly impacts the execution time of short-running queries [43]. To address this issue, Nautilus provides four backends with different low-latency characteristics: a *operator interpreter*, a *byte code interpreter*, *Flounder* [24], and *MIR* [49]. The operator interpreter directly executes Nautilus operators without any tracing, IR generation, or compilation. Thus, it induces no compilation latency ▲▲, but reaches only a very low

throughput ▼▼. It is used during development to simplify the debugging of logic errors in operators. The byte code interpreter, on the other hand, translates the Nautilus IR to a set byte codes as proposed by Kohn et al. [43]. During query execution, it invokes pre-compiled functions for each byte code. This makes it optimal for embedded environments that often allow new code generation at runtime. At runtime, it reaches a higher throughput than the operator interpreter ▼ and only introduces a very low latency for the byte code generation ▲. In contrast, the Flounder [24] backend translates our Nautilus IR directly into machine code using AsmJit [42]. In particular, Flounder is a specialized compiler for data processing workloads and provides a thin abstraction over x64 assembly that performs no additional compiler optimization. This allows Flounder to generate machine code ♦ with negligible compilation times ▲. However, Flounder currently only support x64 platforms and generating machine code for different architectures can require significant engineering effort [28]. Finally, the MIR [49] backend translates our Nautilus IR to the MIR-IR. MIR is a general purpure JIT compiler similar to LLVM, focusing on low compilation times. It was initially developed as a backend for Ruby and provides common compiler optimizations, e. g., dead code elimination, instruction combination, or register allocation. In contrast to Flounder, MIR generates more efficient machine code ▲ with similar compilation times ▲.

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

5.3.2 High-Performance Backends. For long-running queries that process large data sets or streams, it is crucial to maximize execution performance. To this end, Nautilus provides two high-performance code generation backends that aim for optimal code quality: a *MLIR* and a *C++* backend. The MLIR backend translates Nautilus IR to machine code using the MLIR [48] framework. MLIR provides an extensible compiler framework based on LLVM, which is traditionally the most common code-generation framework for compilation-based execution engines [57]. LLVM provides various advanced compiler optimizations, e.g., auto-vectorization, and enables Nautilus to inline proxy functions in the generated code. Furthermore, the MLIR backend can integrate 3rd-party dialects, e.g, LingoDB [37] or Daphne [20], to accelerate specific workloads. As a result, the MLIR backend can generate highly efficient code ▲▲ at a cost of higher compilation latency ♦ (tens of milliseconds). In contrast, the C++ backend translates Nautilus IR to C++ code, which is compiled and linked using a standard compiler at runtime. The generated code is easier to debug and also reaches very high throughput ▲▲. Nonetheless, compiling C++ results in considerable latency ▼ [55].

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

5.3.3 Specialized Backends. Modern data processing workloads often involve UDFs, which cause a high overhead in traditional data processing systems [34, 45]. To this end, Nautilus provides a specialized compilation backends based on Babelfish [30] that accelerates Python, Java, or JavaScript UDFs. This accelerator leverages Truffle [81] and the GraalVM [21] for the execution of pipelines that involve of UDFs and Nautilus operators. Using the Truffle framework, Nautilus implements an bytecode interpreter for the Nautilus IR that integrates with existing language implementations for, e.g., GraalJS [59] and Graal-Python [58]. This enables Nautilus to perform holistic optimization, e.g. inlining and operator fusion, across relational, streaming, and UDF-based operators. As a result, the UDF accelerator eliminates the overhead of UDF-based workloads and can reach a significantly high-performance in comparison to the default backends ▲▲. However, the Graal JIT compiler also introduces a high compilation latency ▼.

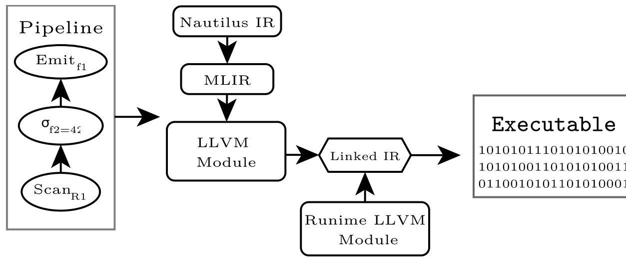


Fig. 6. Inlining of function calls between generated code and runtime of the host system using Nautilus' MLIR backend.

5.4 Optimizations

On the one hand, backends may provide powerful optimizations that lead to highly efficient code. This comes at the price of large dependencies (MLIR backend) or multi-second compilation times (CPP backend). On the other hand, backends may provide very limited optimizations but offer low-latency compilation and introduce no (bytecode interpreter) or small (Flounder) dependencies. Nautilus can optimize the generated IR to improve the performance of these low-latency backends. While these optimizations do not impact the MLIR backend, Nautilus can inline the runtime code of frequent function calls into generated LLVM IR to improve the performance of the MLIR backend.

Constant Folding and Propagation: To only allocate registers for constants if and where they are required, we fold and propagate constants. During tracing, Nautilus creates constants for values that are only known at runtime. The basic block that defines a constant often differs from the basic block that uses the constant. Therefore, constants need to be unnecessarily moved between basic blocks. Additionally, multiple constants with the same value may be defined and used in the same basic block which leads to unnecessary register allocations. In an iterative process, Nautilus first propagates constants to the basic blocks where they are used. Second, Nautilus checks if the constants are actually used and if another constant with the same value can replace them.

Redundant Blocks and Operation Removal: The tracing process introduces redundant basic blocks and operations that Nautilus can remove. After tracing, the generated Nautilus IR often contains basic blocks that only contain a single branch operation. We connect the parent basic blocks of these redundant basic blocks to the child basic blocks and thereby reduce the number of instructions and improve the readability of Nautilus IR. Furthermore, tracing may produce redundant operations. For example, a single basic block may contain multiple load operations that take the same register as input. Performing constant propagation and folding typically increases the number of detectable redundant operations. Removing these operations reduces the number of required registers and instructions.

Runtime Inlining: As discussed in Section 3.2, Nautilus operators, e.g., hash join or aggregations, use function calls to call specific pre-compiled operator logic. Even though these function calls are infrequent, they introduce an overhead and limit compiler optimizations. To mitigate this overhead, Nautilus's MLIR backend enables inlining pre-compiled runtime code as illustrated in Figure 6. For each pipeline, the MLIR backend generates an LLVM IR module, which it links with a pre-compiled set of runtime functions. Subsequently, LLVM performs optimizations on the combined module and produces a single executable. As a result, runtime inlining eliminates the boundary between the generated code and the runtime system. This enables further optimizations, e.g., auto-vectorization, that improve the execution performance but increase compilation latency at the same time.

6 EVALUATION

In this section, we evaluate Nautilus on a diverse set of workloads. First, we introduce our experimental setup in Section 6.1. After that, we conduct two sets of experiments. In Section 6.2, we compare

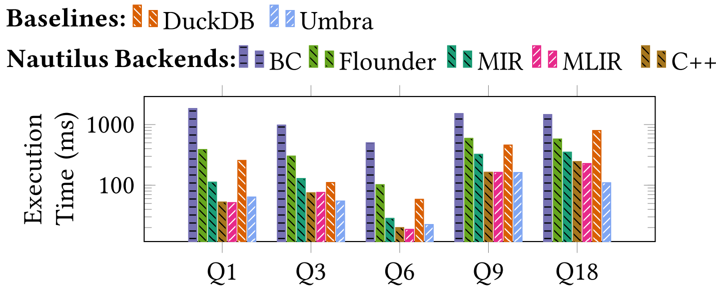


Fig. 7. Query execution time of TPC-H queries 1, 3, 6, 9, and 18 across Nautilus backends, Umbra, and DuckDB (SF1).

the performance of Nautilus across relational, streaming, and UDF-based workloads. In Section 6.3, we perform micro-experiments to study specific aspects of Nautilus.

6.1 Experimental Setup

Throughout our evaluation, we use the following hardware/software configurations and workloads.

Hardware and Software. We execute all experiments on an Intel Xenon Gold 6126 processor with 2.6 GHz and 12 physical cores. Each physical core has a dedicated 32 KB L1 cache for data and instructions. Additionally, each core has 1MB L2 cache, and all cores share a 19.25 MB L3 cache. The test system consists of 755 GB of main memory and runs Ubuntu 22.04. Nautilus' relies on LLVM-16 and GraalVM 22.3. Furthermore, we use Umbra in version 506343a1a and DuckDB version 0.8.1. We execute all measurements using a single thread.

Workloads. Throughout this evaluation, we use the following datasets (stored in main memory in a columnar format). To evaluate the OLAP performance, we use queries from the *TPC-H* benchmark with different scale factors. To assess Nautilus' performance on long-running streaming queries, we use the *Yahoo Streaming Benchmark* [15] and the *NexmarkBenchmark* [76] as representative workloads. For UDF-based queries, we use a set of queries, which was used in multiple previous publications to assess the performance of big data systems on data science workloads [47, 66, 77].

6.2 System Comparison

In this set of experiments, we evaluate Nautilus on relational (see Section 6.2.1), stream processing (see Section 6.2.2), and UDF-based workloads (see Section 6.2.3).

6.2.1 Relational Workloads. In these experiments, we aim to assess the efficiency of Nautilus compilation backends in handling typical relational data processing workloads. We select a subset of queries from the TPC-H benchmark that exhibit a diverse range of relational workload characteristics, such as joins or aggregations. These queries have been used in recent research [38] to evaluate the efficiency of data processing engines. We use Umbra [56], a highly optimized query compilation-based system, and DuckDB [67], a representative vectorized system, as baselines. For the first set of experiments, we keep the data size constant at SF1 and report the execution time for all queries, neglecting the query compilation time. This experiment allows us to evaluate the quality of code generated by various backends. Following this, we vary the data size between SF0.01 and SF10 and report the throughput, expressed in Queries/s, for Q1, Q3, and Q6. This enables us to evaluate the effect of query compilation time and code quality.

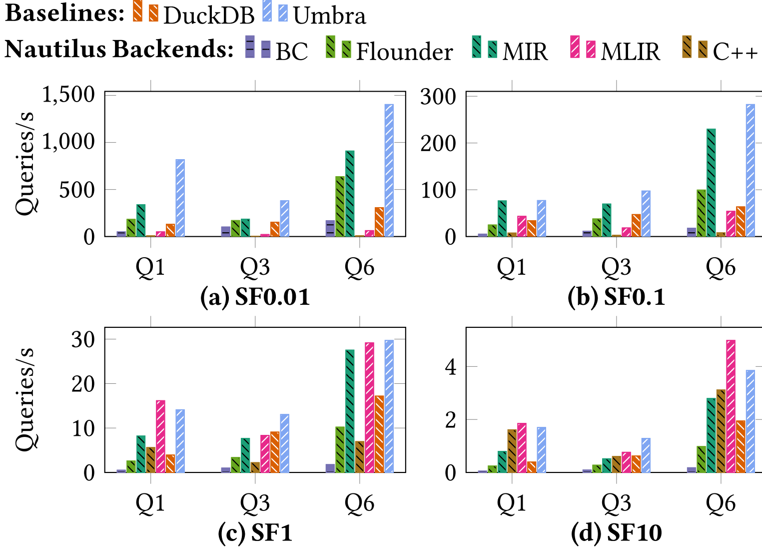


Fig. 8. Comparison of query throughput in queries/s (compilation time + query execution time) on TPC-H queries 1, 3, and 6 across Nautilus backends, Umbra, and DuckDB.

Results. Figure 7 shows the execution time of the selected five queries. The figure demonstrates, as anticipated, that there is a substantial difference in performance (upto 20×) between the high-performance and low-latency backends in Nautilus. The code generated by the high-performance backends in Nautilus is particularly efficient, enabling it to achieve comparable performance to Umbra. However, we can also observe that Umbra outperforms high-performance backends in Nautilus for complex queries, such as Q18. The reason for this is that for such queries Nautilus relies on rudimentary operator implementations, does not have support for group joins, and utilizes a chained hash-table implementation, which results in comparatively inferior performance.

Figure 8 illustrates the throughput of Nautilus' compilation backends, expressed in Queries/s, in relation to Umbra and DuckDB. Across all queries and scale factors, Umbra almost consistently outperforms various backends in Nautilus by adaptively switching from a highly optimized low-latency backed to LLVM in order to generate the most optimal code [43]. It also exhibits superior performance to DuckDB across all queries by up to 6×. For small scale factors (0.01 and 0.1), we observe that Nautilus' Flounder and MIR backends exhibit superior performance compared to it C++ and MLIR backends. This is primarily because higher compilation time constitues the majority of the overall query execution time for C++ and MLIR backends. In addition, we observe that MIR outperforms Flounder by upto 2× as it generates more efficient code with comparable compilation latency. The impact of compilation latency diminishes between scale factors 1 and 10, at which point MLIR reaches the same level of performance as Umbra and surpasses MIR and Flounder. The MLIR backened generates SIMD code for Q6 and SF10, resulting in more efficient code than Umbra. As a result, it outperforms Umbra by a factor of 1.2×. In contrast, Nautilus exhibits inferior performance to Umbra for Q3 owing to the naive chained hash-table implementation which results in several cache misses.

Summary. This experiment shows that the compilation backends of Nautilus can achieve performance - depending on the scale-factor - equivalent to Umbra's highly optimized query compiler. However, no backend is optimal across all scale factors. Thus, an adaptive approach as proposed by Kohn [43] is required to support different workloads efficiently.

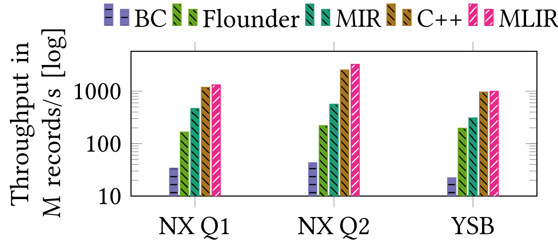


Fig. 9. Comparison on throughput across stream processing queries between different Nautilus backends.

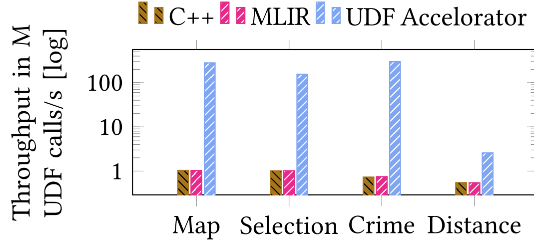


Fig. 10. Comparison on throughput on UDF-based workloads across Nautilus backends and its UDF accelerator.

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

Results. Figure 9 shows the throughput of the executed queries across Nautilus' compilation backends. Across all queries, we observe that Nautilus' high-performance backends, MLIR and C++, achieve the highest performance. Both backends provide sophisticated compiler optimizations, e.g., auto-vectorization or loop-unrolling, to produce efficient code. Interestingly, the MLIR outperforms C++ by up to 1.2x even though both use LLVM as a compiler. This indicates that MLIR, as an intermediate representation, enables additional compiler optimizations compared to C++ code. Furthermore, our results show that Nautilus' low-latency backends are outperformed significantly on long-running workloads.

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

6.2.3 UDF-based Workloads. In this experiment, we evaluate the impact of Nautilus' UDF accelerator across four queries that combine relation operators with one or more Java UDFs with different workloads characteristics. The first two queries, i.e., *map* and *selection*, involve computationally simple UDFs and assess the UDF invocation overhead between the data processing system and the UDF runtime. The third query calculates the average crime index for a set of cities and involves multiple UDFs. In contrast, the last query calculates the distance between two points using Vincenty's formula [78] and is computationally intensive.

Results. Figure 10 shows the throughput of Nautilus' high-performance backends, MLIR and C++, in comparison to its UDF accelerator across all four UDF-based queries. Overall, we can make two key observations. First, the C++ and the MLIR backends achieve the same low performance as the UDF overhead dominates the execution time for both. This involves the invocation of the UDF runtime,

Table 3. Comparison of compilation latency in milliseconds across Nautilus and Umbras compilation backends for TPC-H Queries 1, 3, and 6.

	Nautilus					**Umbra**	
	BC	**MIR**	**Flounder**	**MLIR**	**C++**	**Fast**	**LLVM**
Q1							
Tracing	0.49	0.51	0.50	0.56	1.79	-	-
IR Generation	0.13	0.12	0.16	0.16	0.48	0.94	0.88
Lowering	0.06	0.24	0.17	1.54	0.70	-	-
Code Generation	-	0.85	0.37	24	131	0.32	36.11
Σ Compilation	0.69	1.72	1.18	27.35	134	1.27	36.99
Q3							
Tracing	1.05	1.08	1.07	1.24	4.20	-	-
IR Generation	0.30	0.27	0.26	0.29	1.07	1.55	1.54
Lowering	0.16	0.5	0.04	3.99	1.64	-	-
Code Generation	-	2.1	0.76	59.66	377	0.9	46.63
Σ Compilation	1.51	3.98	2.52	65.16	381	2.45	48.17
Q6							
Tracing	0.19	0.20	0.19	0.23	0.76	-	-
IR Generation	0.04	0.04	0.04	0.04	0.17	0.58	0.58
Lowering	0.03	0.09	0.06	1.13	0.27	-	-
Code Generation	-	0.38	0.11	19.17	122	0.14	14.38
Σ Compilation	0.27	0.74	0.54	20.59	123	0.72	14.96

the data exchange between the host system and the UDF, and the data conversion. Second, the UDF accelerator eliminates this overhead and achieves a speedup of up to two orders of magnitude. In contrast to the Nautilus' high-performance backends, this executes relational operators and UDFs in the same engine, which enables holistic optimizations across operator boundaries. This approach is even beneficial for computational intensive UDF, i.e., the distance query, as it enables further optimization by Babelfish's JIT compiler.

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

6.3 Compilation Backends

In this section, we conduct a set of micro experiments to assess specific aspects of Nautilus. First, we analyze the compilation latency of Nautilus' backends for different queries in Section 6.3.1. Second, we study the impact of the query complexity on the compilation latency to investigate the robustness of individual backends in Section 6.3.2. Finally, we investigate the impact of inlining in Section 6.3.4.

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

Results. Table 3 shows the latency breakdown of Nautilus' compilation backends in comparison to Umbra across the selected TPC-H queries. For Nautilus we observe that its low-latency backends

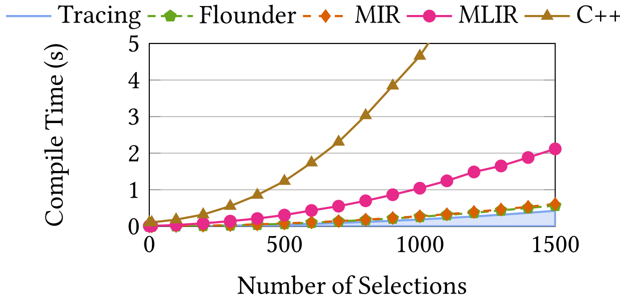


Fig. 11. Compilation time for Tracing, Flounder, MIR, MLIR, and C++ in comparison to the code complexity (number of selections within one operator pipeline).

compile queries in 0.27 to 3.9ms. Among these, Nautilus' bytecode interpreter (BC) reaches the lowest latency as it avoids any code generation. In contrast, Flounder and MIR translate the Nautilus IR to machine code. As MIR performs more optimizations, it requires up to 3x more time to generate code in comparison to Flounder. On the other side of the spectrum, Nautilus' high-performance backends induce a 35x to 100x higher compilation latency. Furthermore, our analysis reveals that Nautilus' tracing approach requires 0.2ms to 1.24ms per query. As a result, it has a high influence on the cumulated compilation time of low-latency backends (up to 60%) but is negligible for high-performance backends. For these, the final code generation dominates the overall compilation time. In case of the C++ backend, compilation time is up to 10x higher than in the MLIR backend. In comparison to Umbra, we can make the following two observations. First, Umbra's low-latency compilation backend achieves compilation times that are, on average, 1.4 times faster than those of Nautilus. These low compilation times are a significant factor for Umbra's superior performance on small scale-factors in the experiments of Section 6.2.1. Second, Umbras LLVM-based backend reaches similar compilation times as Nautilus' MLIR backend. The variations between these two backends can primarily be attributed to different compiler versions and settings.

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

6.3.2 Compilation Latency Robustness. In this experiment, we analyze the impact of the query complexity on the latency of Nautilus' compilation backends. To this end, we assess the compilation latency for queries with 1 to 1500 selections. Each selection increases the control-flow nesting and complexity of the Nautilus-IR.

Results. Figure 10 shows the latency of all compilation backends for an increasing number of selections. As all backends require the initial tracing phase, we indicate the trace latency in blue. The time for tracing increases linearly with the number of selections, as discussed in Section 5.1, and reaches 400ms for 1500 selections. In comparison, Nautilus' low-latency backends only introduce a short additional compilation time. Even for a large number of selections, the latency of Flounder and MIR remains similarly low. In contrast, C++ and MLIR require significantly more time and introduce an overhead of multiple seconds.

Summary. In this experiment, we show that Nautilus' compilation backends scale even to complex queries. However, its high-performance backends introduce a significant compilation latency. To

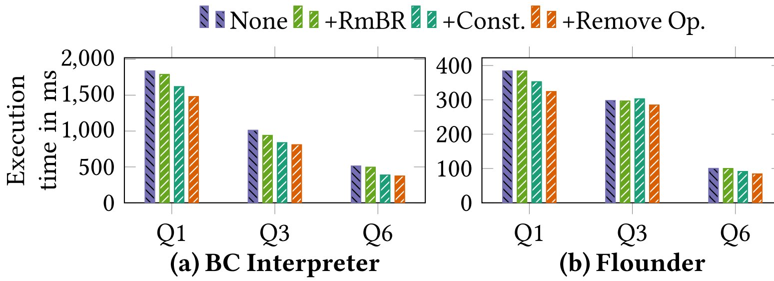


Fig. 12. Impact of Nautilus IR optimizations.

handle such workloads, Nautilus could either rely on its low-latency backends or split complex queries in multiple pipelines, which are easier to compile.

6.3.3 Impact of Nautilus IR optimizations. In this micro experiment, we evaluate the impact of the Nautilus IR-based optimizations introduced in Section 5.4, on the bytecode interpreter and the Flounder backend. These optimizations remove redundant branch-only blocks (RmBR), fold and propagate constants (Const) and remove redundant operations (Remove Op). To assess the impact of these optimizations, we gradually add them and discuss the execution and compilation times for TPC-H Q1, Q3, and Q6 using scale factor 1.

Results. Figure 12 shows the impact of adding the different optimization phases. The RmBR optimization only significantly impacts Q3 for the bytecode interpreter, improving execution time by 7%. The constant folding and propagation and the redundant operation removal optimizations show similar results for Flounder on Q1 and Q6, leading to 8% improvements. The constant optimization is especially effective for the bytecode interpreter, leading to 10%+ improvements across all queries. Overall, the execution times of Flounder are improved by up to 16%. The execution times of the bytecode interpreter are improved by up to 27%. The additional compilation times ranged from 0.1 to 0.5 milliseconds.

Summary. In this experiment, we show that optimization passes on Nautilus IR can significantly impact the execution time of low-latency backends at a relatively small compilation time cost.

6.3.4 Impact of Runtime Inlining. In this micro experiment, we assess the impact of runtime inlining for Nautilus' MLIR backend as proposed in Section 5.4. This optimization eliminates function calls from the generated code, e.g., to access data structures, and enables further compiler optimizations. To assess the impact of this optimization, we report the execution and compilation times of Nautilus for TPC-H Q1, Q3, and Q6 with and without inlining, using scale factor 1.

Results. Figure 13 shows the impact of inlining on the execution time (a) and compilation time (b). For Queries 1 and 3 inlining reduces the execution time by up to 20% as both queries involve function calls to hash-tables that can be inlined. After inlining, the compilation backend can perform further optimizations like loop unrolling, which are not performed in the presents of function calls. In contrast, inlining does not impact the execution time of Query 6 as its generated code contains no function calls. In addition to the execution time, inlining significantly impacts the compilation time, which increases by up to a factor of 2.4x.

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

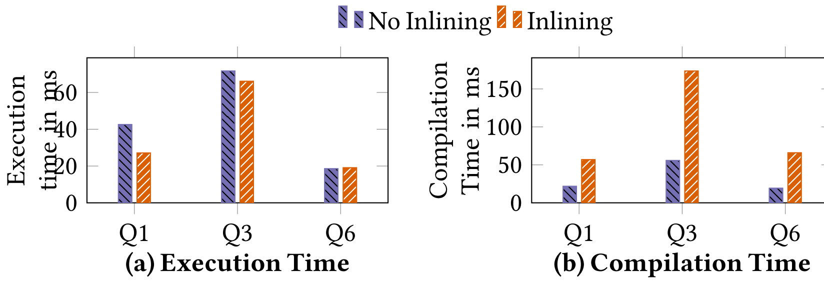


Fig. 13. Impact of runtime inlining for the MLIR backend.

6.4 Discussion

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

7 COMPLEXITY ANALYSIS

Designing and developing a query compiler is inherently challenging; end users building the query engine face an even greater challenge when attempting to use these compilers for daily development tasks. Over the years, we have witnessed widespread use and subsequent discontinuation of query compilers because of such challenges. Our group is involved in a multi-year effort to create a novel data processing system for batch and stream processing called NebulaStream [82, 83]. Over the years, we have developed multiple query compilers for NebulaStream [29, 30] to target different use-cases. Following a discussion on the implementation complexity of various query compilers, this section further describes our personal experience developing multiple query compilers and how it has (i) influenced the design of the Nautilus, and (ii) garnered positive feedback from a small community.

Implementation Complexity. Nautilus' significantly reduces the complexity of compilation-based execution engines. To demonstrate this aspect, we analyses the implementations of representative compilation-based systems. Table 4 compares the lines of code of selected operators implementations across Nautilus, Hawk [11], Mutable [33], MxDB [24], LingoDB [37], and DBLAB [70]. Hawk, Mutable, and MxDB follow a traditional query compilation approach [55]. Each operator provides a template to directly generate low-level code, i.e., C++ in Hawk, WebAssembly in Mutable, and Flounder-IR in MxDB. Depending on the operator, these templates become very large, complexity, and hard to maintain. In contrast, LingoDB and DBLAB use declarative IRs to reduce the complexity of operators. However, this also spreads implementation details across multiple abstraction levels, which makes operators hard to debug. In contrast, Nautilus provides an imperative operator interface and uses tracing to generate efficient code. This enables Nautilus to reduce the code complexity using high-level abstraction that are still easy to debug with common programming tools.

Community Experience. NebulaStream has a small community of contributors comprising bachelor, master, and Ph.D. students with varying degrees of expertise. Over the years, they have engaged with the query compilers either directly, by implementing or improving operators [52, 69], or indirectly, by debugging queries to evaluate specific optimizations [13, 14]. While using the earlier versions of the query compiler most of them needed hands-on supervision in daily tasks, which was

Table 4. Comparison of complexity of query compilers.

	Scan	Projection	Selection	Aggregation	Join
Use Code Templates					
Hawk [11]	115	98	130	389	625
Mutable [33]	108	161	111	1441	242
MxDB [24]	51	/	45	613	216
Use Declarative-IRs					
LingoDB [37]	36	/	14	384	149
DBLAB [70]	22	16	13	83	42
Nautilus	18	16	8	91	190

extremely challenging to scale. Developing an operator implementation interface that is intuitive for users was one solution, which led us to design our interface similar to the widely-adopted Volcano [27] model. This garnered very positive feedback and led to general supervision (e.g., GitHub PR comments). Designing the query compiler, which makes it relatively easier to debug the system, was more challenging. Bugs can persist in operator logic, one of the backends, or in-between (e.g., Nautilus-IR). Implementing a unified IR using our trace-based technique has proven beneficial in these instances. In case a bug persists across multiple backends, it typically indicates that the bug is in the operator logic. In such cases, the interpretation backend greatly simplifies debugging by direct inspection with debuggers such as *gdb*. Nautilus has also helped identify and isolate backend bugs. In multiple instances, we were able to narrow down the source of the bug to a single backend using Nautilus as it exclusively occurred in that backend. Lastly, there is a loose-coupling between Nautilus-IR and different backends, which makes bugs related to undefined behaviour harder to detect. In such cases, we have relied on using *sanitizers* in our compilation backends.

8 RELATED WORK

Over the past decade, query compilation has been the subject of extensive research [46, 55, 68] and implemented in a wide range of data processing systems [1, 23, 46, 51, 55, 56, 61, 62, 80]. In general, Nautilus differs from prior works in three directions, i.e., work on *code generation abstractions*, work on *low-latency query compilation*, and work that applies query compilation to *diverse workloads*.

Code generation abstractions. The first line of research proposed interfaces and abstractions to reduce the complexity of compilation-based query execution engines [2, 4, 11]. Many query compilers generate code in programming languages like C++, Java, or OpenCL to make the generated code easy to debug [11]. However, this often leads to significant compilation latencies. For example, Amazon Redshift employed a global cache to mitigate compilation overhead [4]. However, Nautilus decouples the implementation of operators and their execution. Engineers can debug operators directly while the compiler translates them to efficient code at runtime. Similarly, researchers proposed Domain Specific Languages (DSLs), like LMS [70], Voila [31], Weld [60, 61], or Voodoo [65], to decouple implementation and execution. These DSLs introduce declarative primitives that represent specific data processing operations, e.g., `hash`, `bucket_insert` in Voila [31]. In contrast to Nautilus operators, these DSLs do not map directly to imperative implementations, which hinders testing and debugging. Additional work proposed programming interfaces for code generation [32, 39]. Although this hides the details of code generation, it does not address the fundamental mismatch between the implementation of operators that generate code and the executed code at runtime. In contrast, Nautilus' operators and data structures correspond directly to C++ code, simplifying development, testing, and maintenance. Recent work uses MLIR [48] as a framework for query compilation in data processing systems [20, 36, 37]. These approaches introduce specialized MLIR dialects to model primitive data processing operations similar to Voila [31]. In contrast, Nautilus uses MLIR as a compilation backend for Nautilus IR. To

this end, it only relies on standard dialects, which simplifies our MLIR integration. Nautilus hides the details of MLIR compiler internals from engineers and allows to implement operators in standard C++.

Low-latency Query Compilation. The second research area focused on techniques to reduce the latency of query compilers [24, 32, 39, 43]. To this end, Kohn et al. [43] proposed bytecode interpretation, Funke et al. [24] and Kersten et al. [39] proposed special purpose compilers, and Haffner et al. [32] proposed to use V8 as a JIT-compiler. Nautilus incorporates these approaches and provides different compilation backends using bytecode interpretation, special-purpose compilers, and low-latency JIT compilers. This enables Nautilus to compare the individual techniques and to target a wide range of workloads, including short-running queries. To this end, Nautilus can choose a specific compilation backend depending on the workload and hardware characteristics. This allows Nautilus to support x64 as well as ARM CPU architectures and relieves engineers from the complexity of developing query compilers from scratch. Orthogonal to our work, Wagner et al. [79] recently proposed the generation of a vectorized interpretation-based engine from a query compiler. This is a compelling approach to reduce startup latency and can be combined with the Nautilus operator implementation framework.

Diverse Workloads. The third line of research leverages query compilation to accelerate different data processing workloads, e.g., stream processing [7, 29, 35, 75], spatial data processing [73], machine learning [20], and polyglot queries involving UDFs [17, 22, 30, 71, 72]. Nautilus integrates many aspects of these works to target diverse workloads. It adapts the efficient operators for stream processing proposed by LightSaber [75], Grizzly [29], and Darwin [7] and provides additional compiler optimizations to increase execution performance, e. g., runtime inlining. Furthermore, Nautilus' Babelfish [30]-based UDF accelerator extends previous work like YeSQL [22] and Tuplex [72] and enables holistic optimization across relational operators and UDFs. Supporting these workloads underpins the flexibility of Nautilus' compilation approach.

9 CONCLUSION

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

Overall, Nautilus makes query compilation more accessible to a broader audience. To this end, we plan to provide Nautilus as an open framework that is easy to integrate with different data processing systems. This reduces the engineering effort, which is needed to develop compilation-based execution engines and helps researchers to focus on data-processing related challenges.

10 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 (BIFOLD22B and BIFOLD23B). Furthermore, we thank Matthis Gördel for his support and feedback.

REFERENCES

[1] 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].

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

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*. ACM, 1383–1394. https://doi.org/10.1145/2723372.2742797

[4] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-Invented. In *SIGMOD*. ACM, 2205–2217. https://doi.org/10.1145/3514221.3526045

[5] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System. In *PLDI*. ACM, 1–12. https://doi.org/10.1145/349299.349303

[6] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD*. ACM, 2326–2339. https://doi.org/10.1145/3514221.3526054

[7] Lawrence Benson and Tilmann Rabl. 2022. Darwin: Scale-in stream processing. In *CIDR*. https://www.cidrdb.org/cidr2022/papers/p34-benson.pdf

[8] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In *ICOOOLPS*. ACM, 18–25. https://doi.org/10.1145/1565824.1565827

[9] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*. 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[10] Ajay Brahmakshatriya and Saman Amarasinghe. 2021. BuildIt: A Type-Based Multi-stage Programming Framework for Code Generation in C++. In *CGO*. https://doi.org/10.1109/CGO51591.2021.9370333

[11] Sebastian Breß, Bastian Köcher, Henning Funke, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2018. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal* 27 (2018), 797–822. https://doi.org/10.1007/s00778-018-0512-y

[12] Tom Britton, Lisa Jeng, Graham Carver, Tomer Katzenellenbogen, and Paul Cheak. 2020. Reversible Debugging Software "Quantify the time and cost saved using reversible debuggers". (11 2020).

[13] Xenofon Chatziliadis, Eleni Tzirita Zacharatou, Alphan Eracar, Steffen Zeuch, and Volker Markl. 2024. Efficient Placement of Decomposable Aggregation Functions for Stream Processing over Large Geo-Distributed Topologies. *Proceedings of the VLDB Endowment* 17, 6 (2024), 1501–1514.

[14] Ankit Chaudhary, Steffen Zeuch, Volker Markl, and Jeyhun Karimov. 2023. Incremental Stream Query Merging. In *EDBT 2023*. OpenProceedings.org, 604–617. https://doi.org/10.48786/edbt.2023.51

[15] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Peng, and Paul Poulosky. 2016. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *IPDPS*. IEEE, 1789–1792. https://doi.org/10.1109/IPDPSW.2016.138

[16] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. 2015. An Architecture for Compiling UDF-Centric Workflows. *PVLDB* 8, 12 (aug 2015), 1466–1477. https://doi.org/10.14778/2824032.2824045

[17] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Çetintemel, and Stanley B. Zdonik. 2015. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR*. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper23u.pdf

[18] Andrew Crotty, Alex Galakatos, and Tim Kraska. 2020. Getting Swole: Generating Access-Aware Code with Predicate Pullups. In *IEEE ICDE*. 1273–1284. https://doi.org/10.1109/ICDE48307.2020.00114

[19] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *ACM SIGMOD* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[20] Patrick Damme, Marius Birkenbach, Constantinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina M. Ciorba, Mark Dokter, Pawel Dowgiallo, Ahmed Eleliemy, Christian Faerber, Georgios I. Goumas, Dirk Habich, Niclas Hedam, Marlies Hofer, Wenjun Huang, Kevin Innerebner, Vasileios Karakostas, Roman Kern, Tomaz Kosar, Alexander Krause,

Daniel Krems, Andreas Laber, Wolfgang Lehner, Eric Mier, Marcus Paradies, Bernhard Peischl, Gabrielle Poerwawinata, Stratos Psomadakis, Tilmann Rabl, Piotr Ratuszniak, Pedro Silva, Nikolai Skuppin, Andreas Starzacher, Benjamin Steinwender, Ilin Tolovski, Pinar Tözün, Wojciech Ulatowski, Yuanyuan Wang, Izajasz P. Wrosz, Ales Zamuda, Ce Zhang, and Xiaoxiang Zhu. 2022. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. In *CIDR 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p4-damme.pdf

[21] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *VMIL*. ACM. https://doi.org/10.1145/2542142.2542143

[22] Yannis Foufoulas, Alkis Simitsis, Lefteris Stamatogiannakis, and Yannis Ioannidis. 2022. YeSQL: "You Extend SQL" with Rich and Highly Performant User-Defined Functions in Relational Databases. *Proc. VLDB Endow.* 15, 10 (jun 2022), 2270–2283. https://doi.org/10.14778/3547305.3547328

[23] Craig Freedman, Erik Ismert, and Per-Åke Larson. 2014. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Engineering Bulletin* 37 (2014), 22–30. http://sites.computer.org/debull/A14mar/p22.pdf

[24] Henning Funke, Jan Mühlig, and Jens Teubner. 2020. Efficient Generation of Machine Code for Query Compilers. In *DaMoN*. ACM. https://doi.org/10.1145/3399666.3399925

[25] Henning Funke and Jens Teubner. 2020. Data-parallel query processing on non-uniform data. *PVLDB* 13, 6 (2020), 884–897. https://doi.org/10.14778/3380750.3380758

[26] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. *PLDI* (2009), 465–478. https://doi.org/10.1145/1542476.1542528

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

[28] Ferdinand Gruber, Maximilian Bandle, Alexis Engelke, Thomas Neumann, and Jana Giceva. 2023. Bringing Compiling Databases to RISC Architectures. *PVLDB* 16, 6 (apr 2023), 1222–1234. https://doi.org/10.14778/3583140.3583142

[29] 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, 2487–2503. https://doi.org/10.1145/3318464.3389739

[30] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. 2021. Babelfish: Efficient Execution of Polyglot Queries. *Proc. VLDB Endow.* 15, 2 (oct 2021), 196–210. https://doi.org/10.14778/3489496.3489501

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

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

[33] Immanuel Haffner and Jens Dittrich. 2023. A Simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries. In *Proceedings of the 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28 - March 31, 2023*. OpenProceedings.org.

[34] 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.

[35] Anand Jayarajan, Wei Zhao, Yudi Sun, and Gennady Pekhimenko. 2023. TiLT: A Time-Centric Approach for Stream Query Optimization and Parallelization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. ACM, New York, NY, USA, 818–832. https://doi.org/10.1145/3575693.3575704

[36] Michael Jungmair and Jana Giceva. 2023. Declarative Sub-Operators for Universal Data Processing. *Proc. VLDB Endow.* 16, 11 (aug 2023), 3461–3474. https://doi.org/10.14778/3611479.3611539

[37] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *PVLDB* 15, 11 (jul 2022), 2389–2401. https://doi.org/10.14778/3551793.3551801

[38] 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). https://doi.org/10.14778/3275366.3275370

[39] 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). https://doi.org/10.1007/s00778-020-00643-4

[40] Timo Kersten and Thomas Neumann. 2020. On another level: how to debug compiling query engines. In *Proceedings of the workshop on Testing Database Systems*. 1–6.

[41] Yannis Klonatos, Christoph Koch, Tiark Rompf, and Hassan Chafi. 2014. Building efficient query engines in a high-level language. In *PVLDB*, Vol. 7. VLDB Endowment, 853–864. https://doi.org/10.14778/2732951.2732959

[42] Petr Kobalicek. 2023. AsmJit: Low-Latency Machine Code Generation. https://asmjit.com/. [Online; accessed 22.6.2023].

[43] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive execution of compiled queries. In *ICDE*. IEEE, 197–208.

[44] André Kohn, Viktor Leis, and Thomas Neumann. 2021. Building Advanced SQL Analytics From Low-Level Plan Operators. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1001–1013. https://doi.org/10.1145/3448016.3457288

[45] 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/

[46] Konstantinos Krikellas, Stratis D Viglas, and Marcelo Cintra. 2010. Generating code for holistic query evaluation. In *ICDE*. 613–624.

[47] 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.

[48] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *CGO*. IEEE. https://doi.org/10.1109/cgo51591.2021.9370308

[49] Vladimir Makarov. 2020. MIR: A lightweight JIT compiler project. https://developers.redhat.com/blog/2020/01/20/mir-a-lightweight-jit-compiler-project. [Online; accessed 22.6.2023].

[50] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed Operator Fusion for In-memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last. In *PVLDB*, Vol. 11. VLDB Endowment, 1–13. https://doi.org/10.14778/3151113.3151114

[51] Prashanth Menon, Amadou Ngom, Lin Ma, Todd C. Mowry, and Andrew Pavlo. 2020. Permutable Compiled Queries: Dynamically Adapting Compiled Queries without Recompiling. *PVLDB* (2020). https://doi.org/10.14778/3425879.3425882

[52] Adrian Michalke, Philipp M. Grulich, Clemens Lutz, Steffen Zeuch, and Volker Markl. 2021. An energy-efficient stream join for the Internet of Things. In *DaMoN*. 1–6. https://doi.org/10.1145/3465998.3466005

[53] Josh Mintz. 2017. In this iteration of Database Deep Dives, we had the pleasure of catching up with Professor Andy Pavlo. https://www.ibm.com/cloud/blog/database-deep-dives-with-andy-pavlo

[54] Ingo Müller and otehrs. 2020. The Collection Virtual Machine: An Abstraction for Multi-Frontend Multi-Backend Data Analysis. In *DaMoN*. https://doi.org/10.1145/3399666.3399911

[55] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. In *PVLDB*, Vol. 4. VLDB Endowment, 539–550.

[56] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[57] Thomas Neumann and Guido Moerkotte. 2009. Generating optimal DAG-structured query evaluation plans. *Computer Science-Research and Development* (2009). https://doi.org/10.1007/s00450-009-0061-0

[58] Oracle. 2020. Graal Python. https://github.com/graalvm/graalpython.

[59] Oracle. 2020. GraalJS. https://github.com/graalvm/graaljs.

[60] Shoumik Palkar, James Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *PVLDB* 11, 9 (2018), 1002–1015. https://doi.org/10.14778/3213880.3213890

[61] Shoumik Palkar, James Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, and Matei Zaharia. 2017. Weld: A common runtime for high performance data analytics. In *CIDR*. http://cidrdb.org/cidr2017/papers/p127-palkar-cidr17.pdf

[62] 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].

[63] Mosha Pasumansky and Benjamin Wagner. 2022. Assembling a Query Engine From Spare Parts. In *CDMS*. https://cdmsworkshop.github.io/2022/Proceedings/ShortPapers/Paper1_MoshaPasumansky.pdf

[64] Pedro Pedreira, Orri Erling, Maria Basmanova, Kevin Wilfong, Laith S. Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: Meta's Unified Execution Engine. *PVLDB* 15, 12 (2022), 3372–3384. https://doi.org/10.14778/3554821.3554829

[65] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo - a Vector Algebra for Portable Database Performance on Modern Hardware. In *PVLDB*, Vol. 9. VLDB Endowment, 1707–1718. https://doi.org/10.14778/3007328.3007336

[66] Vignesh Prajapati. 2013. *Big data analytics with R and Hadoop*. Packt Publishing Ltd.

[67] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an embeddable analytical database. In *SIGMOD*. ACM, 1981–1984. https://doi.org/10.1145/3299869.3320212

[68] Jun Rao, Hamid Pirahesh, C Mohan, and Guy Lohman. 2006. Compiled query execution engine using JVM. In *ICDE*. IEEE. https://doi.org/10.1109/ICDE.2006.40

[69] Nils Schubert, Philipp M. Grulich, Steffen Zeuch, and Volker Markl. 2023. Exploiting Access Pattern Characteristics for Join Reordering. In *DaMoN 2023*. https://doi.org/10.1145/3592980.3595304

[70] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to architect a query compiler. In *SIGMOD*. https://doi.org/10.1145/2882903.2915244

[71] Moritz Sichert and Thomas Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *PVLDB* 15, 5 (2022), 1119–1131.

[72] Leonhard Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1718–1731. https://doi.org/10.1145/3448016.3457244

[73] Ruby Y. Tahboub and Tiark Rompf. 2020. Architecting a Query Compiler for Spatial Workloads. In *SIGMOD*. ACM, 2103–2118. https://doi.org/10.1145/3318464.3389701

[74] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General incremental sliding-window aggregation. In *PVLDB*, Vol. 8. VLDB Endowment, 702–713. https://doi.org/10.14778/2752939.2752940

[75] Georgios Theodorakis, Alexandros Koliousis, Peter Pietzuch, and Holger Pirk. 2020. LightSaber: Efficient Window Aggregation on Multi-Core Processors. In *SIGMOD*. ACM, 2505–2521. https://doi.org/10.1145/3318464.3389753

[76] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. *Nexmark-a benchmark for queries over data streams*. Technical Report. Technical Report. Technical report, OGI School of Science & Engineering at .... https://datalab.cs.pdx.edu/niagara/pstream/nexmark.pdf

[77] Shivaram Venkataraman, Zongheng Yang, Davies Liu, Eric Liang, Hossein Falaki, Xiangrui Meng, Reynold Xin, Ali Ghodsi, Michael J. Franklin, Ion Stoica, and Matei Zaharia. 2016. SparkR: Scaling R Programs with Spark. In *SIGMOD*. ACM, 1099–1104. https://doi.org/10.1145/2882903.2903740

[78] Thaddeus Vincenty. 1975. Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations. *Survey review* 23, 176 (1975), 88–93. https://doi.org/10.1179/sre.1975.23.176.88

[79] Benjamin Wagner, Andre Kohn, Peter Boncz, and Viktor Leis. 2024. Incremental Fusion: Unifying Compiled and Vectorized Query Execution. In *ICDE*.

[80] Skye Wanderman-Milne and Nong Li. 2014. Runtime Code Generation in Cloudera Impala. *IEEE Data Engineering Bulletin* (2014). http://sites.computer.org/debull/A14mar/p31.pdf

[81] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-Optimizing Runtime System. In *SPLASH*. ACM. https://doi.org/10.1145/2384716.2384723

[82] 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 for Data and Application Management in the Internet of Things. In *CIDR*. http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf

[83] Steffen Zeuch, Eleni Tzirita Zacharatou, Shuhao Zhang, Xenofon Chatziliadis, Ankit Chaudhary, Bonaventura Del Monte, Dimitrios Giouroukis, Philipp M Grulich, Ariane Ziehn, and Volker Mark. 2020. Nebulastream: Complex analytics beyond the cloud. *Open Journal of Internet Of Things (OJIOT)* 6, 1 (2020), 66–81. https://www.ronpub.com/ojiot/OJIOT_2020v6i1n07_Zeuch.html

Received October 2023; revised January 2024; accepted March 2024