Big Data Analytic Toolkit: A general-purpose, modular, and heterogeneous acceleration toolkit for data analytical engines

Jiang Li, Qi Xie, Yan Ma, Jian Ma, Kunshang Ji, Yizhong Zhang, Chaojun Zhang, Yixiu Chen, Gangsheng Wu, Jie Zhang, Kaidi Yang, Xinyi He, Qiuyang Shen, Yanting Tao, Haiwei Zhao, Penghui Jiao, Chengfei Zhu, David Qian, Cheng Xu[*]
Intel Corporation
{jiang1.li,qi.xie,yan.ma,jian1.ma,kunshang.ji,yizhong.zhang,chaojun.zhang,aia.bdf.moif,david.qian,cheng.a.xu}@intel.com

ABSTRACT

Query compilation and hardware acceleration are important technologies for optimizing the performance of data processing engines. There have been many works on the exploration and adoption of these techniques in recent years. However, a number of engines still refrain from adopting them because of some reasons. One of the common reasons claims that the intricacies of these techniques make engines too complex to maintain. Another major barrier is the lack of widely accepted architectures and libraries of these techniques, which leads to the adoption often starting from scratch with lots of effort. In this paper, we propose Intel Big Data Analytic Toolkit (BDTK), an open-source C++ acceleration toolkit library for analytical data processing engines. BDTK provides lightweight, easy-to-connect, reusable components with interoperable interfaces to support query compilation and hardware accelerators. The query compilation in BDTK leverages vectorized execution and data-centric code generation to achieve high performance. BDTK could be integrated into different engines and helps them to adapt query compilation and hardware accelerators to optimize performance bottlenecks with less engineering effort.

PVLDB Reference Format:
Jiang Li, Qi Xie, Yan Ma, Jian Ma, Kunshang Ji, Yizhong Zhang, Chaojun Zhang, Yixiu Chen, Gangsheng Wu, Jie Zhang, Kaidi Yang, Xinyi He, Qiuyang Shen, Yanting Tao, Haiwei Zhao, Penghui Jiao, Chengfei Zhu, David Qian, Cheng Xu. Big Data Analytic Toolkit: A general-purpose, modular, and heterogeneous acceleration toolkit for data analytical engines. PVLDB, 16(12): 3702 - 3714, 2023.
doi:10.14778/3611540.3611558

PVLDB Artifact Availability:
Intel Big Data Analytic Toolkit is an open-source project and the code is available at https://github.com/intel/BDTK.

1 INTRODUCTION

With the increase in data volume and diversity of analytical workloads, many data processing engines were developed over the years for high-performance data processing under various scenarios and requirements. The huge demand for engine performance is driving unparalleled investments in software (e.g., vectorization[5], query compilation[21], etc.) and hardware (e.g., GPUs, FPGAs, DSPs, and various accelerators). In recent years, the industry[1, 4, 6, 19, 29] and academia[14–16, 23] have made many explorations to use the query compilation and hardware acceleration techniques for the performance improvement. However, adapting these techniques still is a non-trivial task because of the lack of widely accepted architectures and libraries[15, 18, 26]. As a result, engine developers may inevitably build them from scratch and interact with low-level compiler frameworks, hardware drivers, etc., which leads to many duplicate works and unnecessary costs.

The similar issue also affects the vectorized execution. There are lots of vectorized execution engines that were developed in decades, and most of them usually need to reinvent solutions for basic components and features such as operators, expression evaluation, runtime components, SIMD instructions, etc. As a way to address this problem, some reusable, extensible vectorized execution engines[9, 24, 25] were proposed in recent years. These engines are available to be connected to different data processing systems and play a role as the execution backend, providing the same optimization features to different systems. The rise of such modular query processing components means developers can create a new data processing system by composing and extending the pre-built components, rather than starting from scratch or modifying existing systems laboriously.

Inspired by such reusable components, we address the above issue by proposing a lightweight, easy-to-connect, reusable C++ acceleration toolkit library named Intel Big Data Analytic Toolkit (BDTK) to provide optimization techniques like native query plan compilation, expression compilation, and hardware accelerators support for analytical execution engines. With interoperable interfaces, BDTK is available to be integrated into different engines to reduce the effort of leveraging these techniques, which allows developers to focus on some more specialized requirements. For example, integrating the BDTK into the interpretation-based engines for leveraging the benefits of the interpretation-based and the compilation-based execution at the same time. We carefully design and implement the BDTK with the following key considerations:

Lightweight, easy-to-connect implementation. BDTK is designed as a toolkit for more flexible usage compared with the modular execution engines, especially when integrated with some existing systems. Because of fewer dependencies on the runtime components (e.g., task scheduler), it could be integrated into target systems with low invasion, rather than replacing the entire execution engine of the target systems, which is sometimes unacceptable.

[*]Cheng Xu is the corresponding author, cheng.a.xu@intel.com

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611558

Interoperable representation of data and plan. BDTK uses the Apache Arrow[10] columnar format as the standard data format of the interfaces, a widely used in-memory data representation format. Apache Arrow project defines language-independent data formats to support efficient analytic operations on modern hardware and provide good interoperability between different analytic engines. Similar to the Apache Arrow, Substrait[28] project aims to create a well-defined specification for data processing operations and provide good interoperability between different systems, engines, and libraries. With the Apache Arrow and Substrait, BDTK is expected to achieve good compatibility and reusability with most execution engines.

Query plan compiler with good performance. BDTK generates executable code for the plan fragments and expressions at runtime by compiler infrastructure frameworks like LLVM[12]. The compilation-based execution allows the BDTK to keep the flexibility to combine the vectorized execution and the data-centric code generation for better performance. We also created JITLib, a library that provides unified code generation interfaces and better abstraction based on the compiler frameworks, to reduce the development efforts of the query compilation. The details are discussed in subsection 2.3.

Hardware accelerator support. BDTK offloads some heavy workloads to hardware accelerators for better performance and saving computation resources. For example, based on Intel Codec Library (ICL), BDTK achieves good performance improvement in data decompression by leveraging some hardware accelerators such as Intel QuickAssist Technology (QAT). The details are discussed in section 5.

We are attempting to integrate the BDTK with multiple data processing engines such as Velox[24]. In our experiments, BDTK demonstrates significant data processing performance improvement in some scenarios. To the best of our knowledge, BDTK is the first reusable execution engine acceleration toolkit library that provides native query compilation, expression compilation, and hardware acceleration support. It is open-sourced and we anticipate it could be a helpful choice for the developers.

In the rest of this paper, we first briefly introduce the projects and technologies that BDTK depends on, then present the overall architecture of the BDTK and the inside implementation of the query compilation and hardware accelerator support. We discuss the optimization effect of the BDTK by some micro-benchmarks and provide performance results compared with other systems like Velox. Finally, we discuss some open questions and future works of the BDTK.

2 BACKGROUND

2.1 Apache Arrow

Apache Arrow[10] is a popular development platform for in-memory data analytics. It provides a collection of technologies that help big data systems process and exchange data efficiently. It specifies a standardized language-independent columnar memory format for flat and hierarchical data representation. The format has some important features[11]:

- Adjacency of data for sequential access.
- Random access with constant time complexity.
- SIMD and vectorization-friendly.
- Enabling true zero-copy access in shared memory.

Arrow columnar format is very popular and compatible with many analytic engines. BDTK uses the Apache Arrow C data interface as the standard data protocol. Arrow C data interface defines a very small, stable set of C definitions that could be easily copied into any project and used for columnar data exchange in the Arrow format. Some data analytic frameworks already use the Arrow format as the default data representation format or provide utility for the conversion between their internal data format and the Arrow format, which provides good interoperability.

2.2 Substrait

Substrait[28] is a new project that aims to create a well-defined, cross-language specification for data computation operations. It primarily consists of a formal specification, a human-readable text representation, and a compact cross-language binary representation. One of the typical use cases is serializing a query plan fragment and keeping consistent semantics in different query engines (e.g. Apache Spark[32] plan in Trino[13] or Velox[24]).

The structured query language (SQL) does not provide enough detailed information for query execution and is represented in a human-readable format, making it difficult for execution engines to directly execute queries based on SQL. Therefore, most analytical engines usually parse SQL into query plan trees as some internal representations for further optimization and execution firstly, such as the logical plan and physical plan in Apache Spark. These internal plan representations are specialized for different engines, which leads to incompatibility between different data processing systems. With a primary motivation to resolve this issue, Substrait was created to provide a standard, open representation format for logical plans and physical plans and keep the consistency of data computation operations between different data processing systems. For example, the *Join* relation operation takes two inputs fields named left and right, which is represented in the human-readable text representation (JSON) of Substrait as:

```
"join": {
    "left": {
        "filter": {
            "input": {
                "read": { ... },
                ...
            },
        },
    },
    "right": { ... }
}
```

As proven by the practice, the Substrait representation makes good interoperability between different systems effectively, which is adopted by more and more data processing systems such as the Velox[24], DuckDB, and DataFusion, etc. BDTK uses the Substrait as a standard interface to achieve good compatibility with different engines.

2.3 Query Compilation

Because of the rapid increase of main memory capacity, the performance of modern data processing engines is dominated by memory access and CPU usage instead of disk access. The traditional volcano-style iterator execution model, which performs well with disk-based engines, has significant interpretation overhead and works ineffectively with modern hardware[17].

Listing 1: Example query to demonstrate the execution models.

```
SELECT f1(a),f2(a,b) FROM table WHERE a=c;
```

To reduce the interpretation overhead, the vectorized execution model and query compilation were developed over the years. Both of the methods are widely applied to a number of execution engines. Listing 2 and Listing 3 demonstrate the execution procedures of the methods respectively based on the query shown in Listing 1, which contains a filter operator ($a = c$) and a projection operator ($f1(a), f2(a, b)$).

Listing 2: Example of the vectorized execution.

```
vector<int> a_col = ...;
vector<int> b_col = ...;
vector<int> c_col = ...;

vector<int> selected_index = ...; # Filter
eq_int(a_col, c_col, selected_index);

vector<int> output_col1 = ...; # Projection
vector<int> output_col2 = ...;
f1_vec(a_col, output_col1, selected_index);
f2_vec(a_col, b_col, output_col2, selected_index);
```

The vectorized execution model uses pull-based iteration, in which operators produce results by a *next()*-like interface. For example, the *next()* call of the root operator of a query tree will call the *next()* interfaces of its precursor operators recursively to fetch the results until reaching leaf operators. Each *next()* call of the vectorized model produces a vector containing multiple result tuples rather than a single one, which amortizes the interpretation overhead such as virtual function calls. The operations in the vectorized model are performed by a number of type-specialized primitives (like *eq_int()*, *f1_vec()*, and *f2_vec()* in Listing 2), usually implemented as the tight loops execute simple operations(e.g., *eq()*, *fun1()*, and *fun2()*) for multiple tuples. Such implementation allows the vectorized model to take advantage of modern CPUs easily, like the SIMD instructions, out-of-order execution, and deep instructions pipeline[20].

The query compilation eliminates the interpretation overhead by generating efficient executable code for given query plans at runtime. The data-centric code generation[21] is a popular query compilation method and is widely applied in multiple execution engines[14, 16, 23]. In this method, each operator typically has interfaces like *produce()* and *consume()*, which generate the corresponding code for data processing. The code will be compiled into executable machine code via some compiler frameworks like LLVM[12]. The data-centric code generation procedure could be seen as a depth-first traverse of query plan trees, where *produce()* is called on the first visit to collect operators in the same pipeline and *consume()* on the second visit to generate the code of the plan nodes[17]. Pipelines are the plan fragments that match the *source/intermediate/sink* pattern, in which only the sink operator (e.g. *Join* or *Aggregation* operator) needs to materialize the results to memory.

Listing 3: Example of the data-centric code generation.

```
vector<int> a_col = ...;
vector<int> b_col = ...;
vector<int> c_col = ...;

vector<int> output_col1 = ...;
vector<int> output_col2 = ...;
int output_index = 0;
for (int index=0; index<a_col.size(); ++index) {
    int a = a_col[index];
    int b = b_col[index];
    int c = c_col[index];
    # Filter
    if (a == c) {
        # Projection
        output_col1[output_index] = f1(a);
        output_col2[output_index] = f2(a,b);
    }
}
```

Different from the vectorized execution, the operator boundaries in a pipeline are broken in the data-centric code generation, and the operators are fused in a single tight loop as the demonstration in Listing 3. This gives the data-centric code generation more opportunities for performance optimization, such as primitive function inline, loop fusion, instruction combination, etc., which reduces instruction number and allows data to stay at registers as long as possible to minimize memory access[21]. Despite the fact that query compilation has a number of advanced benefits, some engines still refrain from adapting it due to its complexity (e.g., interaction with underlying compiler frameworks), which makes the development, maintenance, and profiling challenging[3]. Some recent query compilation engines[15, 18] tackle this problem by introducing well-designed and implemented abstraction layers between the relational operators and underlying compiler frameworks, which improves extensibility and reduced the complexity significantly.

Some engines[19] that based on the vectorized execution use the data-centric code generation partially to improve the performance bottlenecks. Conversely, some compilation-based engines introduce the vectorized execution partially[16, 20] or generate code with the vectorized-sytle[27] to take advantages like SIMD instructions simultaneously. We believe the query compilation is superior because it leverages the vectorized execution and the data-centric code generation cleverly to achieve better execution performance. On the other head, with proper abstraction layers and tools, the building of a query compiler can become more tractable[22].

3 OVERVIEW OF BDTK

In this section, we introduce the usage of the BDTK and describe how BDTK is integrated with typical execution engines.

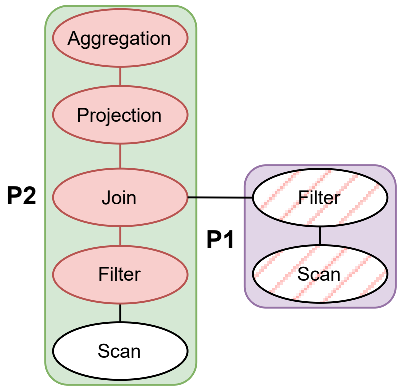

Figure 1: An illustration of the plan fragment optimization using the BDTK. The query tree consists of two plan fragments that use the BDTK in different ways respectively. The use of BDTK is indicated by the red color.

3.1 Overview

BDTK is a reusable, lightweight acceleration toolkit library for analytical execution engines. It provides query compilation, expression compilation, and hardware accelerator support for engines to optimize some performance bottlenecks. BDTK is designed and implemented using a lightweight paradigm to allow easy integration with various engines. Furthermore, to achieve good reusability, BDTK uses the Apache Arrow[10] and the Subtriait[28] to represent the data and the physical query plan fragments respectively, which provides good interoperability with different engines.

Most of the execution engines from the industry prefer to use the interpreted vectorized execution model in recent years, which is easier to develop, profile, debug, and adapt to various data[3]. Nevertheless, some of these engines also implemented the query compilation because of the advantage of cleverly combining the query compilation and the vectorized execution[17, 27]. For example, Clickhouse developed CHJIT[19] to improve the performance of the expression evaluation. BDTK provides a solution for engines to leverage the advantages of query compilation and hardware accelerators easily and avoid building from scratch, which will reduce the development effort significantly.

As shown in Figure 1, a query plan consists of two plan fragments $P1$ and $P2$, and each plan fragment utilizes the BDTK in different ways respectively. BDTK fuses all the operators in the $P2$ as a compound operator and generates an executable function at runtime to boost the data processing performance by leveraging the vectorized execution and the data-centric code generation. In addition to the query plan compilation, some components in the BDTK are available to be embedded into the original execution without replacing the operators. For instance, the *Filter* and *Scan* in the $P1$ apply the expression evaluator and hardware accelerator library of the BDTK respectively.

3.2 Integration with Execution Engines

A typical integration and execution of the BDTK are illustrated in Figure 2. The applications submit a query (e.g., SQL, DataFrame) to the data processing system, where the query is transformed into the logical plans and the physical plans by the parser and the optimizer respectively. The physical plan consists of multiple plan fragments as pipelines, which are executed in order to satisfy the data dependencies. Then, the execution engine analyzes the plan and creates execution tasks for each plan fragment. Finally, the engine submits the tasks to the executor for scheduling and execution.

To compile and execute the selected plan fragment, the engine transforms the plan fragment to the Substrait representation and generates the corresponding `BatchProcessor` by the BDTK. The `BatchProcessor` is a unified execution kernel, which is responsible for the code generation, compilation, and execution of a plan fragment. The processor provides interfaces like *processNextBatch()* and *getResult()* to input data and get results batch-by-batch. It could be embedded into the task execution easily by extending new compound operators and replacing pipeline operators with them. For relational algebra operators, there are two types of the processors such as stateless ones (like *Filter*, *Projection*) and stateful ones (like *Join*, *Aggregation*). The stateful processors need to maintain the data processing states (e.g., hash tables) and export the results from the states in the Arrow format.

In addition to the query compilation, BDTK also provides hardware accelerator support to offload some CPU-intensive tasks to specific accelerators, which can reduce the CPU workload and improve the overall performance. For example, BDTK provides a library to support high-performance compression/decompression for the *Scan* operation by leveraging the Intel QAT accelerator, which is discussed in section 5.

The construction of the processor depends on several components in the BDTK, such as the Substrait plan parser, codegen translator planner, code generation toolkit, hardware accelerator library, and memory management. The details of them are discussed in the following sections.

4 QUERY COMPILATION IN THE BDTK

As shown in Figure 2, the basic workflow of BDTK's query compilation is:

- Firstly, the plan fragment in the Substrait representation is parsed by the Substrait plan parser to an internal representation structure. The structure describes detailed information about the plan fragment such as input data types, operators, and expressions (e.g., projection targets, filter conditions, group-by keys, aggregation values, etc.).
- Based on the structure, the codegen translator planner generates a translator list to generate code of the plan fragment via a *Produce/Consume*-like[21] pattern. The translators are responsible for generating code for different operators (e.g., *Filter* translator, *Projection* translator).
- Next, the translator list is triggered in the batch processor builder and generates code (e.g., LLVM IR) for different operators and expressions by using a code generation toolkit library named JITLib. The toolkit library provides unified code generation interfaces and better abstraction between the relational algebra operations and the compiler frameworks.

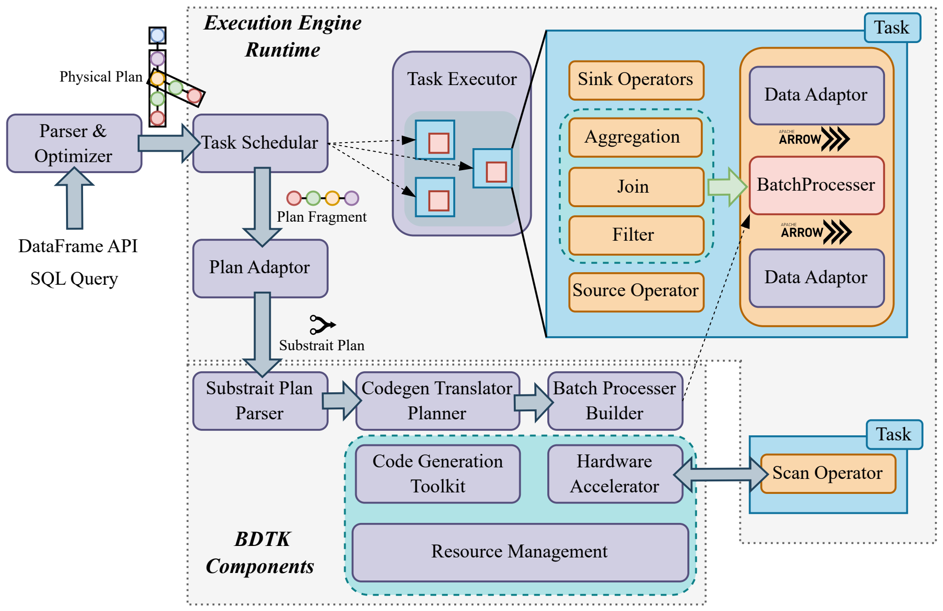

Figure 2: An illustration of how the BDTK integrates with a typical query execution engine. BDTK builds `BatchProcessors` based on the given query plan fragments and embeds them into the execution tasks for data processing.

- Finally, the generated code is optimized and compiled to an executable function, which is used by the batch processor for data processing.

The details of the components related are described in the following sections.

4.1 Substrait Plan Parser

The internal plan fragment representation structure in the BDTK comes from [29] and consists of operator fields. Each operator field contains some expression translators, which are organized as tree structures and used for expression code generation (e.g., filter condition). The Substrait plan parser first translates the expressions in the given Substrait plan fragment into the expression translators, then constructs an internal representation structure by grouping the expression translators to the corresponding operator fields. For example, a binary expression translator represents a binary operation between two expressions typically including:

- **Op Type:** indicates the operation type of the binary expression, including logical operations, arithmetic operations, comparison operations, etc.
- **Left Right Operands:** indicates the argument translators to generate the left and the right operands respectively.
- **SQL Type:** defines the return type of the expression (e.g., `Bool` for the comparison operation).

Currently, the supported expression translations from the Substrait to the BDTK are described in Table 1.

The Substrait plan parser should validate the given plan fragments before the translation, to ensure all of the functions in the plan fragments are supported in the BDTK. The plan parser implements a mechanism to check the plan fragments and trigger fallback if the validation failed. When offloading a plan fragment to the BDTK, the execution engines use the plan parser to detect whether the plan fragment is supported. The plan fragment validation mainly includes the following aspects:

- **Function support validation.** The Substrait plan parser leverages the function lookup engine provided by the Substrait project to check the plan fragments. The function information of the different execution engines and the BDTK are all defined in YAML files respectively. The function signatures used in the given plan fragments are also represented in the Substrait protocol. Based on the function signatures, the detailed information could be retrieved from the corresponding YAML files by the lookup engine. Then, the plan parser searches the YAML file of the BDTK to

Table 1: Mapping between the Substrait representation and the BDTK internal plan representation

Substrait	BDTK
Expression_Literal	ConstantExpr
Expression_Selection	ColumnExpr
Expression_ScalarFunction	UnaryExpr, BinaryExpr, StringOper, etc.
Expression_Cast	CastExpr
Expression_IfThen	CaseExpr
AggregationFunction	AggExpr

get the function with the same semantics and triggers the fallback if there is no function match.

- **Operator support validation.** In addition to the function validation, the plan parser also supports the operator validation. The physical plan fragment contains all execution details such as data types, operator types (e.g., *Join types*), etc. The plan parser validates such information and triggers the fallback if something is unsupported in the BDTK.
- **Rule-based validation.** The plan parser also supports defining some fine-grained rules to validate special cases for some intricate scenarios.

Based on the Substrait plan parser, the given plan fragments from the execution engines are validated and translated to the internal plan representation structures of the BDTK, which will be processed further to generate the executable code.

4.2 Codegen Translator Planner

The codegen translator planner is a component that transforms the internal plan representation structures of the BDTK into the translator lists. The translator list generates the code for the corresponding plan fragments via the *Produce-Consume* paradigm. For each given plan fragment, there are data processing functions defined as *void query_func(RuntimeContext* ctx, ArrowArray* input)* compiled from the generated code. The *ctx* represents a `RuntimeContext` that is responsible for the memory management inside the function, which is described in subsection 4.6.

Firstly, the planner builds the translators of the relational algebra operators according to the fields in the internal structure. For example, the selected plan fragment in Figure 2 will be transformed into a translator list like `FilterTranslator`→ `JoinTranslator` →`AggregationTranslator`. The operator translators generate the code of the internal expressions by the expression translators they contained. The details are discussed in subsection 4.5 and subsection 4.4 respectively.

To make the generated code executable, the planner also inserts some helper translators into the translator list to generate some additional helper code. The planner inserts a special translator named `QueryFuncInitializer` at the head of the list to generate some initialization code that loads the data column from the *input* and allocates output *ArrowArray* in the *ctx*.

In the BDTK, there are two kinds of input/output types for the operator translators, `Row` and `Batch`. It will generate code with the data-centric pattern if both of the adjacent translators have the output and the input as the `Row` type. Based on this way, multiple such successive translators are fused into a single data processing pipeline to reduce the materialization overhead until the last operator translator or the translators with the `Batch` output. If the input type of the pipeline is `Row`, the planner will insert a special translator `ColumnToRowTranslator` at the head of the pipeline that builds the loop control flow and generate data loading code inside it. Similarly, the planner will insert another special translator `RowToColumnTranslator` at the tail of the pipeline to materialize the pipeline results to the memory. After the insertion of these helper translators, the translator list is available to generate the code that will be compiled into an executable *query_func().*

Furthermore, to improve the performance of the generated code, the planner also analyzes operators and expressions inside the pipeline and transforms the translator list. The planner moves some vectorizable expressions from the projectionfilter translators and builds vectorized projectionfilter translators, which generate multiple independent tight loops for facilitating the compiler framework to generate the code optimized with the SIMD instructions (e.g., SSE, AVX2, AVX512) and leveraging the advantages of the modern CPUs. The detailed implementation is discussed in subsection 4.5

The codegen translator planner constructs the translator lists that generate code by leveraging the data-centric code generation and the vectorized execution to achieve good performance. The details of the code generation procedure of the translator lists are discussed in the following sections.

4.3 JITLib

JITLib is a toolkit library for translators to generate the code for data processing at runtime. It provides friendly interfaces to build the data processing code including primitive data types, primitive operations, and control flow.

4.3.1 Motivation. The motivation for developing the JITLib is the gap between the low-level compiler framework and the relational algebra operators. The code representations in these compiler frameworks (e.g., LLVM IR) are similar to the assembly language and have some properties significantly different from the high-level programming languages (e.g., C/C++, Java), such as the single static assignment (SSA)[7], etc. Therefore, it will be very tough to develop the operator and expression translators using the low-level compiler frameworks directly for most of the developers. Shaikka et al.[26] proposed a layered query compiler architecture, which progressively translates the query plans from high-level representation to low-level representation through multiple intermediate representations (IR). The architecture splits the translation process into multiple steps, which reduces the development difficulty and provides more flexibility for optimization. Based on this architecture, JITLib could be seen as one of the IR between the relational algebra operators and the compiler frameworks. It provides a code representation similar to the C language, which not only reduces the effort of the development but also can support new compiler frameworks without modifying the existing translators.

4.3.2 Implementation. JITLib consists of three abstract components as `JITModule`, `JITFunction`, and `JITValue`, each of them defined some unified interfaces and implemented for the different compiler frameworks respectively. Currently, JITLib has provided LLVM-based implementation.

`JITModule` is a compilation unit that contains several `JITFuncti-ons` and provides interfaces like function declaration and compilation. The module is also responsible to manage optimization configurations (e.g., auto-vectorization[31]), and compilation options (e.g., enabling AVX2, AVX512). The LLVM-based implementation of the module leverages a number of compiler passes in the LLVM to optimize the code sufficiently, such as auto-vectorization provided by the `LoopVectorizePass` to apply the SIMD instructions. Besides, the module is available to link with the other modules to import the pre-build or the pre-compiled functions for reducing

```
auto a_col = func->getArgument(0);
# Initializing other JITValue.
...
auto output_index =
  func->createVariable(JITTypeTag::INT32, "output_index", 0);
auto loop_builder = func->createLoopBuilder();
auto index = func->createVariable(JITTypeTag::INT32, "index", 0);
loop_builder->condition([&index, &len]() { return index < len; })
  ->loop([&](LoopBuilder*) {
    auto a = a_col[index];
    auto b = b_col[index];
    auto c = c_col[index];
    auto if_builder = func->createIfBuilder();
    # Filter
    if_builder->condition([&]() { return a == c; })
      ->ifTrue([&]() {
        # Projection: f1(a)
        output_col1[output_index] = func->emitRuntimeFunctionCall(
          "f1",
          JITFunctionEmitDescriptor{.ret_type = JITTypeTag::INT32,
                                    .params_vector = {a}});
        # Projection: f2(a,b)
        output_col2[output_index] = func->emitRuntimeFunctionCall(
          "f2",
          JITFunctionEmitDescriptor{.ret_type = JITTypeTag::INT32,
                                    .params_vector = {a, b}});
        output_index = output_index + 1;
      })
      ->build();
    })
    ->update([&index]() { index = index + 1; })
    ->build();
func->createReturn(output_index);
```

(a) Code generation procedure using JITLib.

```
...
.For_Condition:
  %index.0 = phi i32 [ 0, %.Start ], [ %4, %.Loop_Update ]
  %output_index.0 = phi i32 [ 0, %.Start ], [ %output_index.1, %.Loop_Update ]
  %0 = icmp slt i32 %index.0, %len
  br i1 %0, label %.Loop_Body, label %.After_Loop
.Loop_Body:
  %1 = getelementptr i32, i32* %a_col, i32 %index.0
  %2 = getelementptr i32, i32* %b_col, i32 %index.0
  %3 = getelementptr i32, i32* %c_col, i32 %index.0
  br label %.If_Condition
.Loop_Update:
  %4 = add i32 %index.0, 1
  br label %.For_Condition
.After_Loop:
  ret i32 %output_index.0
.If_Condition:
  %5 = load i32, i32* %3
  %6 = load i32, i32* %1
  %7 = icmp eq i32 %6, %5
  br i1 %7, label %.If_True, label %.If_False
.If_True:
  %8 = load i32, i32* %1
  %9 = call i32 @f1(i32 %8)
  %10 = getelementptr i32, i32* %output_col1, i32 %output_index.0
  store i32 %9, i32* %10
  %11 = load i32, i32* %1
  %12 = load i32, i32* %2
  %13 = call i32 @f2(i32 %11, i32 %12)
  %14 = getelementptr i32, i32* %output_col2, i32 %output_index.0
  store i32 %13, i32* %14
  %15 = add i32 %output_index.0, 1
  br label %.After_If
.If_False:
  br label %.After_If
.After_If:
  %output_index.1 = phi i32 [ %15, %.If_True ], [ %output_index.0, %.If_False ]
  br label %.Loop_Update
```

(b) Generated LLVM IR optimized with the `Mem2Reg` pass.

Figure 3: Example of the code generation procedure for the query *SELECT f1(a),f2(a,b) FROM table WHERE a=c.*

the effort of generating complex operations (e.g., some operations of string, and hash functions).

`JITFunction` includes some code blocks that share the same context, initially containing two blocks: local variable block and start block. The local variable block is used to initialize all of the local variables of the function, it is the first code block to execute to ensure the variables are accessible within the whole function. The start block is executed later and contains the main procedure of the function. The function is responsible for building `JITValue` (variable values and literal values) and maintaining the control flow (the loop and the branch). In the LLVM-based implementation, the functions that are not time-consuming but frequently used should be labeled as `AlwaysInline`. When calling such functions, their definition code will be copied into the module and inline them at the calling point by `AlwaysInlinerPass` for reducing the function calling overhead and better code optimization.

`JITValue` is the basic unit to represent the data and the operations in the JITLib. There are several primitive data types and operations defined as Table 2 shown. Most of the operations are

Table 2: Primitive Types and Operations in JITLib

Types	Operations
Int[8-128]	+, -, *, /, %, =, ≠, <, ≤, >, ≥, static_cast, bitwise
Bool	&&, ||, not, =, ≠, static_cast, bitwise
Float	+, -, *, /, =, ≠, <, ≤, >, ≥, static_cast
Double	+, -, *, /, =, ≠, <, ≤, >, ≥, static_cast
Pointer<T>	+, [], dereference, reinterpret_cast

overloaded as the C++ operators for programming convenience. Each of the operations generates the corresponding code and returns the result as the new `JITValues`, which makes the code generation very clear and comprehensible. For example, $r = v1 + v2$, in which the r, the $v1$, and the $v2$ are all the `JITValues`, generates an `add` instruction of the $v1$ and the $v2$ and produces the result represented by the r. The result values are not assignable in the LLVM-based implementation because of the SSA form. However, this form is awkward to use for building the data processing code in which the values may be assigned multiple times. For this problem, the `JITValue` can be built as a variable that allows overwriting, typically implemented by allocating a piece of memory in the stack to store the value. The LLVM provides the `Mem2Reg` pass to transform such variables to the SSA form efficiently. JITLib also supports the pointer type as a primitive type and related operations like indexing, dereference, and type-casting, which greatly facilitates the load/save operations. Based on the `JITValue` with these primitive types, the data types in the Apache Arrow[10] data format can be represented by composing the values easily.

The implementation of the control flows in the JITLib is based on the *Builder Pattern*, such as `IfBuilder` and `LoopBuilder`. The `IfBuilder` needs three functors to build an *if-else* control flow. The first functor should return a `JITValue` with *Bool* type as the condition, the others are used to generate code in the different branches respectively. The `LoopBuilder` is used to build a *for-loop* control flow. Similar to the `IfBuilder`, the `LoopBuilder` needs a functor to check the loop condition. It also needs two functors to generate code in the loop body and update the condition respectively. Typically, the implementations of these control flows depend on leveraging the code blocks and the jump instructions provided

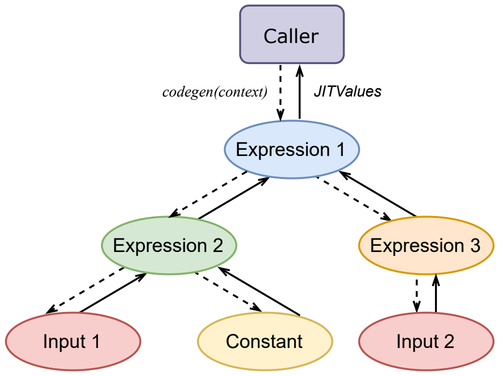

Figure 4: An illustration of the expression evaluation code generation in the BDTK.

by the compiler frameworks, which is very intractable. Based on these builders, the control flow building in the plan fragment code generation is simplified significantly.

4.3.3 Example. To demonstrate the usage of the JITLib more clearly, a code generation procedure is shown in Figure 3 that produces the data processing operations of Listing 3.

By using the JITLib, the code generation procedure shown in Figure 3a is similar to the code in Listing 2 written with C++. The generated LLVM IR is shown in Figure 3b. Based on our practice, the JITLib provides a set of very easy-to-use interfaces to interact with the underlying compiler frameworks, which makes it easier for the developers to build, optimize, and extend the operator and expression translators.

4.4 Expression Evaluation

BDTK builds an expression translator framework to generate the expression evaluation code by using the JITLib. The framework includes various expression types such as arithmetic, logical, comparison, string operations, etc. The code generation procedure of the expression tree is shown in Figure 4. This section describes the implementation details of the expression translators.

Expression evaluation code generation: In Figure 4, the expressions are organized as a tree structure and each of them provides a *codegen()* interface to generate the code. The implementation depends on the JITLib. The root translator will call the interfaces of its child translators recursively and get their results as the arguments. Each expression translator will cache the generated results to reduce the code generation time consumption. The leaves of the trees are the input column expressions or the constant expressions. The values of the input expressions are generated by the `ColumnToRowTranslator`, which builds the loop control flows and generates the data loading code described in subsection 4.2.

Runtime evaluation optimization: The optimization of the expression evaluation in the BDTK mostly depends on the compiler framework when compiling the code at runtime. For example, the common sub-expression elimination and constant folding for the scalar data are applied by the compiler frameworks. Besides, for the data types like the string type that are not handled efficiently by the compiler framework, the optimizations are applied by the BDTK's expression evaluation. The expression translators can identify the sub-expression trees that only have constant inputs, and compute the expression results at the compilation time. Before the code generation, the translator planner traverses the expression trees and merges the same sub-trees to eliminate the computation of the common sub-expressions. For example, the input expressions *a* of the *f1(a)* and *f2(a,b)* in Listing 1 can be merged, which makes the input only needs to be loaded once for each iteration.

Vectorized evaluation propagation: The expression translators generate scalar operations code inside the loop control flows built by the `ColumnToRowTranslator` normally. To transform the scalar operations into vector operations using the SIMD instructions for performance, the expressions need to provide information to the translator planner to construct vectorized operator translators described in subsection 4.2. The expression trees generate the information from the leaves to the root progressively. An expression supporting the vectorized evaluation requires the following conditions: (a) the operations could be auto-vectorized by the compiler frameworks or have self-defined vectorized evaluation functions. (b) all of the children support the vectorized evaluation. Following the rules, each expression in the tree is labeled to indicate whether it supports the vectorized evaluation for the planner.

Null value handling: The arrow format represents null values as the bitmap, which is very efficient for vectorized execution. However, the operations of the bitmap (e.g., loading or storing 1 bit) violate the auto-vectorization when they are mixed with the other operations in a loop. For this problem, the null value computation code is separated into another loop to avoid the violation, which only contains the bitwise operations and takes advantage of the data-centric code generation and the SIMD. The bool type in the arrow format is also represented as the bitmap and also handled in the same way as the null value. The more detailed implementation is described in subsubsection 4.5.1.

External function integration: The expression translators of the complex operations (e.g., regular expressions) require the pre-build or the pre-compiled functions. The operations are defined with C++ and transformed to the LLVM IR by using the Clang compiler. The pre-build functions are transformed to the LLVM IR with the complete definition for the JITLib to inline the functions. The pre-compiled functions are compiled as executable binary and export the declarations as the LLVM IR only, to reduce the time consumption of the query compilation. These methods allow the expression evaluation of BDTK to have good flexibility to extend the functions and achieve good performance.

4.5 Operators

The relational algebra operator supported in the BDTK include *Filter*, *Projection*, *Aggregation*, and *Join*. BDTK provides operator translators to generate the corresponding code for the operators via *Produce-Consume*[21] paradigm. Some important implementation and optimization details of the operator translators are described in this section.

4.5.1 Filter and Projection. `FilterTranslator` and `ProjectTranslator` are used to generate code for the *Filter* and the *Projection* operator respectively. The project translator has the input and the output type as the Row, which contains several expression translators to generate the projection target expressions. Similar to the project translator, the filter translator also has the Row typed input and output. It contains an expression translator to generate the value of the filter condition and an *if-else* control flow. The subsequent translators will generate their code inside the true branch of the control flow.

The basic operator translators described above generate code in the data-centric style, which fuses all operators in the same pipeline into a loop body to keep the data inside the registers as long as possible. However, as the operations become complex, fusing all operations into a single loop may obstruct the application of auto-vectorization. The auto-vectorization prefers to work with the tight loops, as the complex loops are more likely to contain the operations that cannot be vectorized, e.g., function calls and bitwise operations.

To leverage the SIMD instructions by the auto-vectorization, BDTK implements VectorizedProjectTranslator. The translator divides the target expressions into different expression groups and generates independent tight loops for each group respectively. The vectorized translator accepts some vectorizable expression trees from the translator planner. These expression trees are separated into sub-expression trees by the expressions with the bitwise bool-typed inputs or outputs. Commonly, there are three types of sub-expression trees: bool input and bool output (BIBO), normal input and bool output (NIBO), and normal input and normal output (NINO). The sub-expression trees with the same type and shared input expressions are grouped into the same expression group. The vectorized translator generates the tight loops for each group sequentially based on their dependency relationship.

The vectorized translator generates different loops for the different expression group types. The loop for the BIBO is only used for the bitwise bool operations (e.g., logical operations), in which the null value operations and the other bitwise operations can be applied on each byte of the input bool bitmaps to avoid violating the auto-vectorization. The outputs of the NIBO expression trees (e.g., comparison operations) need to be saved into the bool bitmaps from bytes to bits, which is hardly handled by auto-vectorization. Therefore, in the loop of the NIBO, the outputs are first saved as byte arrays. After the loop, the byte arrays are converted to the bitmaps by the SIMD instructions.

4.5.2 Aggregate and Join. BDTK supports *Aggregation* and *Join* operators via HashAggregationTranslator and HashJoinTranslator respectively, both of them have the input type as the Row type. The translators basically generate data processing procedures with the paradigm that combines the runtime-generated code and the pre-compiled hash tables.

The hash aggregation translator contains the group key expression translators and the aggregation target expression translators. For the global aggregation, the translator first generates code that allocates a piece of memory to hold the aggregation results. For the group aggregation, the translator first generates code to pack the group keys and get the corresponding slot by calling the probe interface of the hash table. Then the translator generates the corresponding aggregation functions (e.g., *sum()*, *avg()*, *count()*, *min()*, *max()*, etc.).

For the Join operator, BDTK only supports the equal-inner Join currently, which is implemented by the hash Join. The hash join translator generates code for the join probe side and contains expression translators of the join condition. The translator generates code to pack the key values and get the index of the matched values in the hash table generated by the join build side. Then, the translator builds a loop control flow to load the matched values from the hash table, and the subsequent translators will generate their code inside the join loop body.

Clearly, the performance of the hash tables is very critical for these translators. BDTK implements a set of well-designed hash tables for the join and aggregation. The hash tables are selected by an adaptive strategy that mainly considers the data types, ranges, and cardinality of the keys, which provides good performance for different scenarios. For example, if a hash table has the key type as *int8*, the range of the key will be in $[-128, 127]$ and the maximum size of the table is 256. Therefore, the table allocates 256 slots in advance and the input keys can be mapped to the corresponding slots directly without the hash lookup. For the string-typed hash key, BDTK uses a hash method similar to [33] which designed an efficient hash lookup method.

In addition to the adaptive selection, the hash tables also design some operations to leverage the SIMD instructions. For example, the operations of the key hashing and the key probing both are optimized by the SIMD instructions, which improves the performance significantly.

4.6 Memory Management

The memory in the BDTK consists of the memory used by the host library and the memory used by the runtime-generated code. The overall memory management depends on allocators that are organized as a hierarchical structure. Meanwhile, based on the allocators, the memory in the runtime code is managed by CodegenContext, RuntimeContext, and StringHeap. This section describes the details of these components.

4.6.1 Hierarchical Allocator. Most of the BDTK host objects with small memory consumption are allocated at C++ stack and heap directly. For the data buffers containing lots of items, hash tables for *Join* and *Aggregation*, etc., BDTK provides the allocators to manage their large memory consumption. The allocator has interfaces like *allocate()*, *deallocate()*, *reallocate()*, and *getCapacity()*. The users of BDTK could overload these interfaces and assign the root allocators to the batch processors, which makes it possible to track and manage the memory usage in the BDTK. Each of the objects that use large memory in the BDTK is assigned a child allocator derived from the root, such hierarchical work style is similar to [23].

4.6.2 Context. The memory and objects related to the runtime-generated code are managed by CodegenContext and RuntimeContext. The codegen context is used to maintain the objects represented in the JITValue at the code generation phase, such as the memory buffers for the intermediate results, hash tables, etc. During the code generation phase, the translators register the construction parameters of the required objects into the codegen context, and the context generates the corresponding JITValues to represent the objects. The `JITValues` of the objects can be used to generate related data processing operations, even if the objects are not constructed actually. After the code generation, the objects in the codegen context are constructed based on the registered parameters and managed by the runtime context. The codegen context and runtime context associate the objects in the code generation phase and the execution phase.

4.6.3 String Heap. For the string operations, there are usually some large intermediate results that need to be materialized to the memory and require frequent memory allocations (e.g., the operations like $concat(lower(a), upper(b))$ will generate three intermediate results for each execution). The frequent memory allocation and release lead to significant runtime overheads such as the interaction with the system kernel. BDTK implements `StringHeap` as a part of the runtime context to manage the memory used by the string operations. The string heap uses an arena allocator to manage the memory allocation, which reduces the runtime overheads and improves the performance. To avoid excessive memory usage, the string heap tracks the memory usage and shrinks it at runtime.

5 HARDWARE ACCELERATOR SUPPORT IN BDTK

Data compression and decompression are important for the data processing systems which could reduce the I/O workloads and the costs of storage and network. However, data compression and decompression require many CPU resources and affect the engine performance significantly. In recent years, offloading such CPU-intensive workloads to the specialized hardware accelerators demonstrates significant performance improvement[6]. In the BDTK, we propose Intel Codec Library (ICL) to provide high-performance data compression/decompression by leveraging the hardware accelerators.

ICL is a compression and decompression library, which could be used in some popular data formats, such as Apache Arrow IPC and Apache Parquet. The library is designed as a two-layer architecture as shown in Figure 5.

The upper layer is the bridge layer, which is used to interact with the front-end frameworks and runtimes. For example, it extends the compression class from Hadoop/Spark and forwards the compression requests to the underlying native execution layer through the Java Native Interface (JNI).

The bottom layer is the native execution layer which is a shared C/C++ library to provide accelerated compression and decompression ability. The native execution layer takes full advantage of the hardware accelerators on the Intel Xeon CPUs, such as the Intel QuickAssist Technology (QAT) and the Intel In-Memory Analytics Accelerator (IAA). The ICL not only supports the Intel hardware accelerators to accelerate the deflate-compatible data compression algorithm but also supports some software solutions such as the Igzip from the Intel Intelligent Storage Acceleration Library (ISA-L)[30] that uses the AVX-512 instructions to accelerate the data codec.

Figure 6 demonstrates the compression and the decompression throughput with the dynamic Huffman coding algorithm and the default compression/decompression level 6. For the compression, the QAT could provide a maximum throughput of 124 Gbps for the 32 Kb data pieces. For the decompression, the QAT provides a maximum throughput of 537 Gbps for the 1024 Kb data pieces.

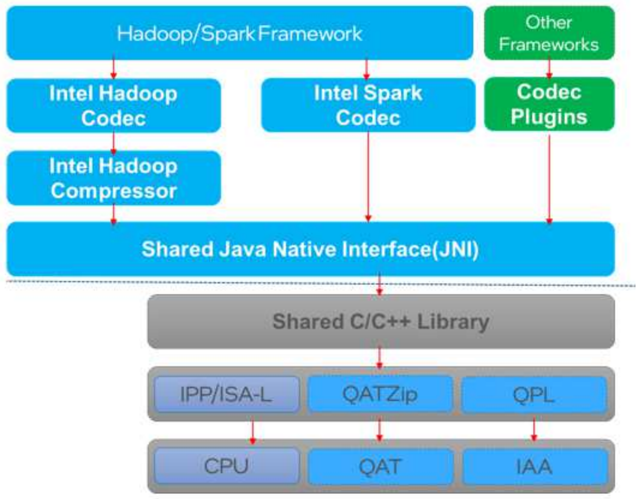

Figure 5: Intel Codec Library Architecture.

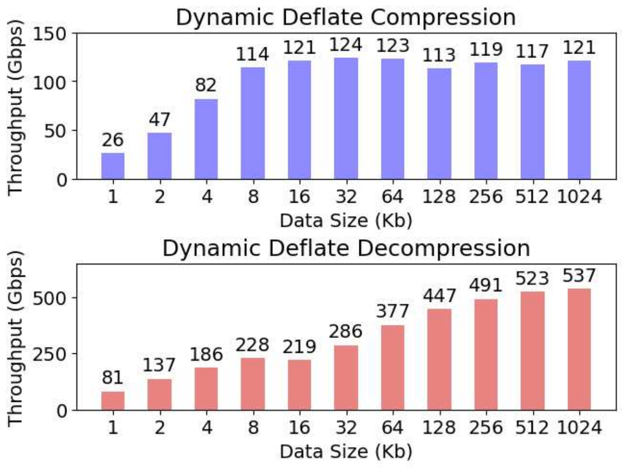

Figure 6: The compression (Upper) and decompression (Lower) throughput of the QAT.

Parquet is a popular and widely used columnar file format for data analytical engines, which supports data compression/decompression storage. To improve the data compression/decompression performance, the ICL is integrated with the Parquet and boosts the performance by leveraging the QAT accelerator and the ISA-L library.

6 EXPERIMENTS

In this section, we present some experimental evaluation results to demonstrate the improvement of the query compilation and the hardware accelerators in the BDTK. We performed the evaluation on the Aliyun Cloud Service with the ECS nodes equipped with

192 vCPUs provided by Intel Xeon Platinum 8475B Sapphire Rapids CPUs, and 512 GB of DRAM.

6.1 Query Compilation Performance Evaluation

We first integrated the BDTK with the Velox[24] to evaluate the performance improvement by using the BDTK. We performed a micro-benchmark to evaluate the expression evaluation performance and an end-to-end benchmark to evaluate the overall query processing performance.

Table 3: Expression Evaluation Cases ($a \in int16; b \in int32; d, e, f, g \in bool$)

Case Id	Expression
Case 1	$a \times a \times a \times a$
Case 2	$b \times b$
Case 3	$a \times a \times 2 + a/3 - 1$
Case 4	$d\ AND\ e$
Case 5	$(f\ OR\ g)\ AND\ ((f\ AND\ (f \neq (f\ OR\ g))))\ OR\ (d = e)$

We designed some expression evaluation benchmark cases as shown in Table 3. Case 1, case 2, and case 3 are the basic arithmetic expressions, in which case 3 is more complex relatively. Similarly, case 4 and case 5 are the basic logical expressions and case 4 consists of multiple logical operations.

In the micro-benchmark of the expression evaluation, we compared the BDTK with Velox's vectorized expression evaluation engine. The benchmark data is generated by the vector fuzzer provided by Velox. The data batch size is 10000 and about 50% of the data are null values. We enabled the AVX2 for both the BDTK and Velox to leverage 256-bit SIMD instructions. BDTK uses LLVM-9 as the query compilation backend. The evaluation results are demonstrated in Figure 7.

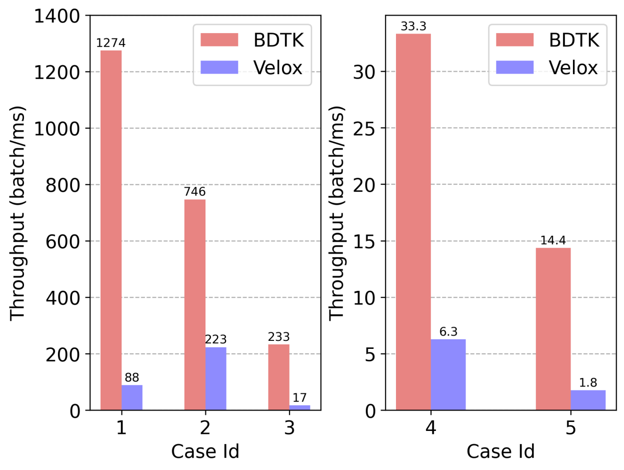

Figure 7: Evaluation results of the expression evaluation in the BDTK and Velox. Left: arithmetic expressions; Right: logical expressions (Higher is better).

Figure 7 demonstrates the throughput of the expression evaluation in the BDTK and Velox. For case 1, case 2 (simple arithmetic expression), and case 4 (simple logical expression), BDTK shows 14x, 3x, and 5x performance improvement respectively. For the complex ones like case 3 and case 5, BDTK is nearly 10x faster than Velox. The primary reason is the expression compilation in the BDTK leverages both the vectorized execution and the data-centric code generation, which takes advantage of the runtime compilation and the SIMD instructions. Compared with the BDTK, the expression evaluation engine of the Velox has some extra overheads like the interpretation cost and materialization cost because of the interpreted vectorized execution. BDTK is available to eliminate the interpretation cost and allows the intermediate results to stay in the registers as long as possible to reduce the memory access by the native compilation. Meanwhile, BDTK can provide more performance improvement for complex expressions. One of the additional costs of the BDTK is the compilation latency, for most common expressions and plan fragments, the compilation latency of BDTK using the LLVM backend is in the two-digit millisecond range, which is relatively trivial, especially for long-running queries.

Furthermore, we evaluated the end-to-end performance improvement by using the Presto engine. The Presto provides an implementation named Prestissimo that uses Velox as the native execution engine. We compared the query execution time of the original Prestissiom and the Prestissiom integrated with the BDTK. We built some queries that select the expression cases in Table 3 as the project target. We used the Hive query runner provided by Prestissimo that creates the tables of the TPC-H benchmark and launches the Hive service. The end-to-end benchmark uses the lineitem table of the TPC-H benchmark with a scale factor of 1000.

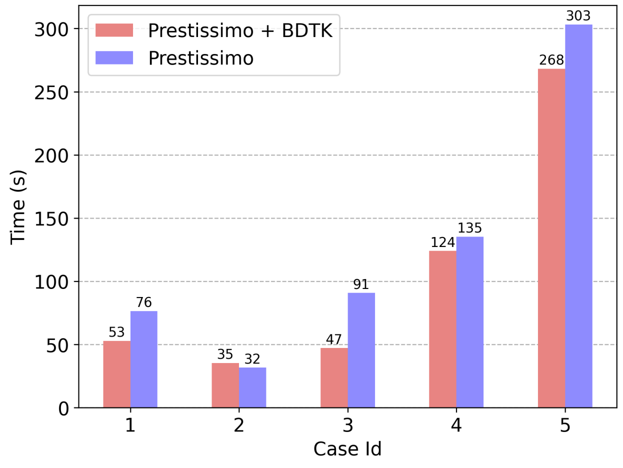

Figure 8: Evaluation results of the end-to-end query execution performance, in which Prestissimo is the Presto engine using Velox as the execution backend (Lower is better).

The results are demonstrated in Figure 8. The results show BDTK provides up to 1.21x performance improvement than the original Prestissimo. However, the improvement is not as significant as the micro-benchmark shown. After the profiling and investigation, we noticed that the conversion between the Velox vector and the Arrow data format leads to lots of redundant memory allocation, which damages the end-to-end performance. This is because the current conversion implementation does not reuse the memory properly.

6.2 Data Decompression Performance Evaluation

We applied the BDTK's hardware accelerator into the native Parquet reader in Velox and improved the data decompression performance, which is critical to the table scan operations. To evaluate the performance improvement provided by the QAT accelerators on the Sapphire Rapids CPU and the Igzip library, we used the TPC-H lineitem table with a scale factor of 100 and measured the decompression time consumption.

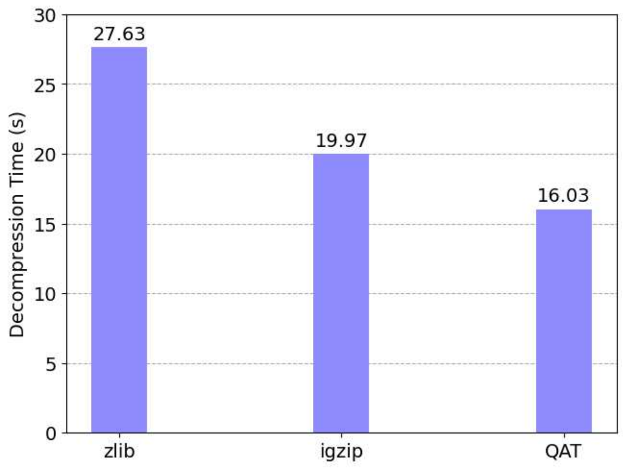

Figure 9: Decompression time consumption under different codec libraries. (Lower is better).

We evaluated the Parquet reader performance with different codec libraries, such as the zlib, the Igzip, and the QAT. We executed the decompression by a single thread multiple times and measured the average decompression time to reduce the variance. As Figure 9 shown, for the single-threaded case, there are 1.72x and 1.38x performance improvements compared with the zlib implementation by using the QAT hardware accelerator and the Igzip library uses the AVX-512 instruction set. We also evaluated the performance with multiple threads in parallel, as both the Igzip and QAT demonstrate good performance improvements and scalability within several threads. Besides, a limited impact of compression ratio was observed after adopting both the QAT accelerator and the Igzip.

7 RELATED WORKS

Apache Arrow[10] provides Gandiva[8] to generate code at runtime for expression evaluation performance. Gandiva consists of a runtime expression compiler based on the LLVM[12], and a high-performance execution environment. The users submit an expression tree, then Gandiva compiles the tree to a high-performance function kernel to consume data in the Apache Arrow format and produce results. Gandiva also combines the vectorized execution with the compilation to leverage the SIMD instructions. Compared with the query compilation in the BDTK, the usage scope of Gandiva is restricted to the expression evaluation.

Velox[24] is a unified execution engine developed by Meta. Velox aims to provide reusable, extensible, high-performance data processing components, which could improve engineering efficiency and democratize optimizations. Although there are several design concepts of Velox shared with BDTK like reusability, Velox is a complete data processing engine that includes the task, task scheduler, task executor, etc. By contrast, BDTK is a lightweight library and focuses on providing high-performance utilities for the engines, rather than replacement.

MapD[29] is a compilation-based database management system (DBMS) leveraging GPU for data processing acceleration. The code generation in MapD is templated-based[2], which is inflexible[26] and difficult to optimize compared with the *Produce/Consume*[21] pattern used in BDTK. MapD generates code of data processing as LLVM IR, then compiles it to native CPU code by LLVM JIT compiler or to native GPU code through NVVM IR.

Hyper[16] is a DBMS pioneered data-centric code generation[21] based on LLVM. Hyper fuses all operators in a pipeline as a compact loop to improve code locality and reduce materialization cost, in which their boundaries are blurred and tuple attributes could be kept in CPU registers as long as possible. BDTK not only leverages data-centric code generation but also takes advantage of vectorized execution to boost performance further. BDTK focuses on providing reusable, easy-to-connect utilities for engines instead of a complete DBMS like Hyper.

8 CONCLUSION

As the important techniques for improving query performance, query compilation and hardware accelerators are explored and adapted in many engines. However, because of the intricacies of these techniques, adapting these techniques often has to start from scratch and interact with low-level compiler frameworks, hardware drivers, etc., which leads to many duplicate works and unnecessary costs.

To address this problem, we proposed BDTK, a reusable C++ acceleration toolkit library with interoperable interfaces, which could be integrated into different engines and help them to adapt high-performance data processing utilities. In this paper, we described the implementation details of the BDTK, which explains how BDTK achieves reusability and good performance. We integrated the BDTK into the Velox and evaluated the performance of expression evaluation and data decompression, in which BDTK showed significant performance improvements.

As future steps, we are extending the features and optimizing the performance of BDTK. There are several open questions in BDTK to resolve, such as the expansibility for expressions and functions, the observability for debugging and profiling, etc. We are also attempting to integrate the BDTK with multiple data processing engines for more practical feedback. Finally, we hope the BDTK could be a helpful choice for the developers to easily leverage the query compilation and the hardware accelerator techniques.

REFERENCES

[1] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *Proceedings of the 2022 International Conference on Management of Data*. 2205–2217.

[2] Morton M. Astrahan, Mike W. Blasgen, Donald D. Chamberlin, Kapali P. Eswaran, Jim N Gray, Patricia P. Griffiths, W Frank King, Raymond A. Lorie, Paul R. McJones, James W. Mehl, et al. 1976. System R: Relational approach to database management. *ACM Transactions on Database Systems (TODS)* 1, 2 (1976), 97–137.

[3] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, et al. 2022. Photon: A fast query engine for lakehouse systems. In *Proceedings of the 2022 International Conference on Management of Data*. 2326–2339.

[4] BlazingDB. 2023. Home - SQL-Bblaz Ing. https://blazingsql.com/. (Accessed on 02/23/2023).

[5] Peter A Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *Cidr*, Vol. 5. 225–237.

[6] Monica Chiosa, Fabio Maschi, Ingo Müller, Gustavo Alonso, and Norman May. 2022. Hardware acceleration of compression and encryption in SAP HANA. In *48th International Conference on Very Large Databases (VLDB 2022)*.

[7] Ron Cytron, Jeanne Ferrante, Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1991), 451–490.

[8] Apache Software Foundation. 2018. Gandiva: A LLVM-based Analytical Expression Compiler for Apache Arrow | Apache Arrow. https://arrow.apache.org/blog/2018/12/05/gandiva-donation/. (Accessed on 02/26/2023).

[9] Apache Software Foundation. 2022. Apache Arrow DataFusion — Arrow DataFusion documentation. https://arrow.apache.org/datafusion/. (Accessed on 02/23/2023).

[10] Apache Software Foundation. 2023. Apache Arrow | Apache Arrow. https://arrow.apache.org/. (Accessed on 02/23/2023).

[11] Apache Software Foundation. 2023. Arrow Columnar Format — Apache Arrow v11.0.0. https://arrow.apache.org/docs/format/Columnar.html. (Accessed on 02/23/2023).

[12] LLVM Foundation. 2023. The LLVM Compiler Infrastructure Project. https://llvm.org/. (Accessed on 02/23/2023).

[13] Trino Software Foundation. 2023. Trino | Distributed SQL query engine for big data. https://trino.io/. (Accessed on 02/23/2023).

[14] Carnegie Mellon University Database Group. 2023. NoisePage – Self-Driving Database Management System. https://noise.page/. (Accessed on 02/23/2023).

[15] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an open framework for query optimization and compilation. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2389–2401.

[16] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 195–206.

[17] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proceedings of the VLDB Endowment* 11, 13 (2018), 2209–2222.

[18] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *The VLDB Journal* 30 (2021), 883–905.

[19] Maksim Kita. 2022. JIT in ClickHouse. https://clickhouse.com/blog/clickhouse-just-in-time-compiler-jit. (Accessed on 02/23/2023).

[20] 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. *Proceedings of the VLDB Endowment* 11, 1 (2017), 1–13.

[21] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[22] Thomas Neumann. 2021. Evolution of a compiling query engine. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3207–3210.

[23] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*.

[24] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3372–3384.

[25] Pola-rs. 2023. Polars. https://www.pola.rs/. (Accessed on 02/23/2023).

[26] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to architect a query compiler. In *Proceedings of the 2016 International Conference on Management of Data*. 1907–1922.

[27] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. 2011. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. 33–40.

[28] Substrait. 2023. Home - Substrait: Cross-Language Serialization for Relational Algebra. https://substrait.io/. (Accessed on 02/23/2023).

[29] Alex Suhan and Todd Mostak. 2015. MapD: Massive throughput database queries with LLVM on GPUs.

[30] Gregory Tucker, Roy Oursler, and Johnathan Stern. 2017. ISA-L igzip: improvements to a fast deflate. In *2017 Data Compression Conference (DCC)*. IEEE Computer Society, 465–465.

[31] Michael Joseph Wolfe. 1995. *High performance compilers for parallel computing*. Addison-Wesley Longman Publishing Co., Inc.

[32] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[33] Tianqi Zheng, Zhibin Zhang, and Xueqi Cheng. 2020. Saha: a string adaptive hash table for analytical databases. *Applied Sciences* 10, 6 (2020), 1915.