Analyzing and Optimizing Java Code Generation for Apache Spark Query Plan

Kazuaki Ishizaki
IBM Research
ishizaki@jp.ibm.com

ABSTRACT

Big data processing frameworks have received attention because of the importance of high performance computation. They are expected to quickly process a huge amount of data in memory with a simple programming model in a cluster. Apache Spark is becoming one of the most popular frameworks. Several studies have analyzed Spark programs and optimized their performance. Recent versions of Spark generate optimized Java code from a Spark program, but few research works have analyzed and improved such generated code to achieve better performance. Here, two types of problems were analyzed by inspecting generated code, namely, access to column-oriented storage and to a primitive-type array. The resulting performance issues in the generated code and were analyzed, and optimizations that can eliminate inefficient code were devised to solve the issues. The proposed optimizations were then implemented for Spark. Experimental results with the optimizations on a cluster of five Intel machines indicated performance improvement by up to 1.4× for TPC-H queries and by up to 1.4× for machine-learning programs. These optimizations have since been integrated into the release version of Apache Spark 2.3.

CCS CONCEPTS

• **Computing methodologies** → *Distributed programming languages*; • **Software and its engineering** → *Dynamic compilers.*

KEYWORDS

Apache Spark; Code generation; Optimization

ACM Reference Format:
Kazuaki Ishizaki. 2019. Analyzing and Optimizing Java Code Generation for Apache Spark Query Plan. In *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE '19), April 7–11, 2019, Mumbai, India.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3297663.3310300

1 INTRODUCTION

Distributed parallel processing on a large cluster is widely used to process huge volumes of data. The Pioneer frameworks of this processing approach were Google MapReduce [15] and Apache Hadoop [3], which offer simple programming models with only `map()` and `reduce()` operations. These MapReduce frameworks enable easy development of data-intensive applications such as those for data analytics and relational data processing. In addition to them, more flexible distributed in-memory computing frameworks [25, 39] have also been developed.

Apache Spark [48] is a well-known distributed in-memory computing framework. Spark is written in Java and Scala, which run on a Java virtual machine (JVM). Spark has three advantages over traditional MapReduce frameworks. The first is that Spark keeps as much intermediate data in memory as possible to accelerate computation and it provides a functional and declarative API for abstraction of immutable distributed in-memory data. The second advantage is that such abstraction enables the use of any memory storage format including column-oriented storage [13], which is known as an efficient format for keeping consecutive fields of a column within adjacent memory addresses. This is because relational data processing can be accelerated by vectorization and because data caching requires a smaller memory size at a high compression ratio. From Spark 1.5, Project Tungsten started to use a custom binary representation [46], called the *Tungsten's representation* here, to alleviate the overhead of JVM memory management [30]. The final advantage is that Spark can put a set of operations into a single compilation scope [32] when it can generate optimized Java code from a given program. This can leverage advanced compiler optimizations within a large compilation unit.

Spark uses the above technologies together. As their mixture constitutes a new approach, Spark has new and unique workload characteristics. There have been several studies on characterizing and tuning Spark workloads at the operating system and JVM levels [12, 38], file system level [10], or CPU-architecture level [7], and on analyzing and improving performance bottlenecks by applying block-time analysis [33]. Essertel et al. [16] showed performance issues for accessing and decoding data with an in-memory data representation by using sampling profiling. The following two performance issues were complementarily identified by carefully inspecting the generated code from a Spark program. To the best of my knowledge, this is the first paper to describe optimizations at the generated-code level in Spark.

Listing 1: Example of a Spark program

```
val ds: Dataset[Array[Double]] = Seq(Array(0.5, 0.6), Array(1.5, 1.6)).toDS.
    cache
for (int i = 0; i < 2; i++) {
  val ds1 = ds.filter(a => a(0) > i).map(a => a)
  ds1.show
}

// output  i = 0: (0.5, 0.6), (1.5, 1.6)
//         i = 1: (1.5, 1.6)
```

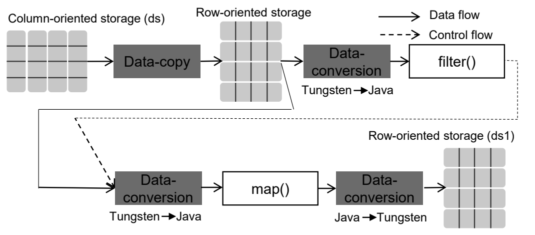


Figure 1: Overview of code generated from Listing 1

1. **Data-copy** occurs between two different physical memory layouts. When column-oriented storage [6] is used, the generated code always executes data-copy from column-oriented to row-oriented storage or vice versa.
2. **Data-conversion** occurs between two different logical representations. When a program that accesses primitive-type arrays is executed, the generated code includes data-conversions between the Java-data and Tungsten's representations through boxing and unboxing.

Listing 1 shows an example of a Spark program embedded in Scala. Line 1 creates a `Dataset` that includes two rows. Each row has a double primitive-type array. This `Dataset` is stored to an in-memory cache implemented via column-oriented storage. Then, at line 3, if the first array element in a given row is greater than `i`, a `map()` function returns a pointer to the input array. Line 4 then prints the elements of an array in the `Dataset`. The Spark runtime generates Java code from the operations in line 3. Then, the generated Java code is translated into native code by a just-in-time (JIT) compiler in a JVM, and the native code for line 3 is executed. Figure 1 gives an overview of the Java code generated by Spark from this program. The `Data-copy` operation copies data from column-oriented to row-oriented storage. The `Data-conversion` operation then converts data between different representations. If a condition in `filter()` is satisfied, `Data-conversion` executes additional conversion to the Java-data representation and one conversion from the Java representation to the Tungsten's representation. These operations in red are newly generated by the Spark runtime, which degrades performance. These issues are discussed further in Sections 3.1 and 4.1.

Inspecting code generated by Spark showed that such data copying and conversion was generally unnecessary. This paper generalizes the data-copy problem as an optimization problem for efficient access to column-oriented storage in runtime-generated code with an iterator-based loop. In addition, it generalizes the data-conversion problem as an optimization problem to handle two-valued logic (i.e., a primitive-type array) in a Spark runtime originally designed for three-valued logic (i.e., values in SQL), and to exchange data between JVM and framework-managed memory regions that a Java or Scala program cannot interpret. These are common problems because column-oriented storage and three-valued logic are also used in other big data processing frameworks [25]. Three optimizations were thus devised to eliminate unnecessary operations for these two problems. The optimizations were implemented for Apache Spark and experimental results using benchmark programs showed the efficiency of the proposed optimizations. The optimizations have since been integrated into the release version of Apache Spark 2.3 with refinements [20–23].

This paper makes the following contributions.

- Identifying performance issues by inspecting generated code in Spark (see Sections 3.1 and 4.1).
- Devising an optimization to eliminate data-copy for accessing column-oriented storage (see Section 3.2).
- Devising two optimizations to eliminate data-conversions for accessing a primitive-type array (see Section 4.2).
- Showing the following performance improvements (reduction of elapsed time) on a cluster with five Intel Xeon machines (see Section 5) by
  - up to 1.41× (geometric mean of 1.10×) for 22 TPC-H queries; and
  - up to 1.42× (geometric mean of 1.21×) for two machine-learning algorithms.

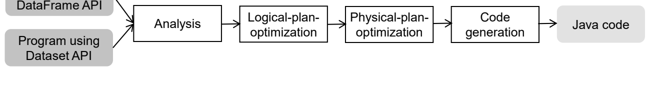
2 APACHE SPARK

This section gives an overview of a novel optimizer in Apache Spark and two of Spark's programming APIs, DataFrame and Dataset for an embedded domain-specific language (DSL).

2.1 Optimizer

Spark provides a functional programming API for abstraction of immutable distributed in-memory collections called *resilient distributed datasets (RDDs)* [47]. A program using the RDD API is executed without any optimizations for the whole program. Spark also has a novel optimizer called *Catalyst* [6], which can optimize operations in a program by using information on their schema with the DataFrame API or on their types with the Dataset API. These APIs are explained in Section 2.2.

Figure 2 shows the phases in Catalyst, analysis, logical-plan-optimization, physical-plan-optimization, and code generation. The analysis phase creates a tree form for a logical plan. Each node in the tree corresponds to a data source or



Program using DataFrame API
Program using Dataset API
Analysis
Logical-plan-optimization
Physical-plan-optimization
Code generation
Java code

Figure 2: Overview of the Catalyst optimizer

computation in a program using the DataFrame or Dataset API.

The logical-plan-optimization phase applies optimizations among multiple operations such as Boolean expression simplification, predicate push-down, and cast simplification. The physical-plan-optimization phase then generates operations that are executed on the Spark execution engine. Spark is written in Java and Scala, and runs on a JVM. This phase applies an optimization to determine whether multiple operations can be put into one loop instead of using *volcano style* [18], which passes data by using an iterator between two operations. This model has the efficiency advantages of pipeline processing.

Finally, the code-generation phase generates Java code from one or multiple physical operations. When the physical-plan optimization phase determines the appropriateness of putting multiple physical operations into one loop, the code-generation phase generates Java code for the multiple operations in the loop to avoid generating the iterator. This expands the optimization scope of the JIT compiler in a JVM [32], and is called *whole-stage code generation.* To improve performance, the generated code operates on data in Tungsten's representation instead of Java's data representation.

When it is invoked, the generated code repeats a loop iteration to process each row. The loop includes the following steps:

(1) **Read row**: Read data for a row in the Tungsten's representation.
(2) **Read element**: Read an element for each column from the data read in step 1.
(3) **Convert to Java data format**: Convert data from Tungsten's representation to Java-data representation used in step 4.
(4) **Perform computation**: Perform computation by using Java operations.
(5) **Convert to Spark data format**: Convert data from the Java-data representation to the Tungsten's representation used in steps 6 and 7.
(6) **Write element**: Write an element for each column in a row.
(7) **Write row**: Write data to a row in the Tungsten's representation.

Project Tungsten [46] includes three optimizations: (1) optimized code generation, (2) custom memory management and binary processing, and (3) cache-aware computation. As mentioned above, optimization 1 is referred to as *whole-stage code generation.* Optimization 2 uses a custom binary representation to reduce the number of indirect references and the number of Java objects to represent core data structures such as a row or column; this is referred to as the *Tungsten's representation.*

Listing 2: Example program using the DataFrame API

```
1 // df is stored in column-oriented storage
2 val df: DataFrame[Double] = Seq(0.5, 1.5).toDF("x").cache
3 for (i <- 0 to 1) {
4   val df1 = df.filter("x >" + i).selectExpr("x * 2 AS v")
5   df1.show
6 }
7 // output  i = 0: 1.0, 3.0
8 //         i = 1: 3.0
```

Listing 3: Logical and physical plans for Listing 2

```
== Logical Plan ==
Project [(x * 2.0) AS v]
+- Filter (x > 0.0)
   +- InMemRelation [x]
== Physical Plan ==
*Project [(x * 2.0) AS v]
   +- *Filter (x > 0.0)
      +- InMemTblScan [x]
         +- InMemRelation [x]
```

Listing 4: Pseudo Java code generated for line 4 in Listing 2

```
final class GeneratedIterator {
  Iterator inputIterator = ...;
  Row projectRow = new Row(1);
  RowWriter rowWriter = new RowWriter(projectRow);
  protected void processNext() {
    while (inputIterator.hasNext()) {
      // (1) Read a row
      Row inputRow = (Row) inputIterator.next();
      // (2) Read an element
      // (3) Convert to Java data format
      double x = inputRow.getDouble(0);
      // (4) Perform computation
      if (!(x > 0)) continue;        // filter("x > 0")
      double value = x * 2;          // selectExpr("x * 2")
      // (5) Convert to Spark data format
      // (6) Write an element
      rowWriter.write(0, value);
      // (7) Write a row
      appendRow(projectRow);
    }
  }
}
```

2.2 Programming APIs

Since version 1.3, Spark has provided a declarative API, called the *DataFrame API*, for an abstraction of immutable distributed rows with a schema [6]. This abstraction, called a *DataFrame*, is like a table in a relational database. Since version 1.6, Spark has also provided a type-safe, object-oriented programming API, called the *Dataset API* [5]. The Dataset API enables programmers to use a lambda expression in Scala or Java.

2.2.1 DataFrame API.

A DataFrame can execute relational operations and keep track of its schema and operations. Listing 2 shows an example of a Spark program using the DataFrame API. Line 2 creates a DataFrame `df` that includes two rows. Each row has a double value `0.5` or `1.5` in column `x`. The program executes the statement at line 4 twice with two different values of `i`. The data in the `df` at line 2 is stored to an in-memory cache by a `cache` operation. Building the cache prevents creating multiple instances of `df` in the loop. This is useful for iterative algorithms that are commonly used in machine learning. At line 4, when `i = 0`, the value `0.5` from column `x` is first read. Because the condition `x > 0` in the `filter()` operation is satisfied, `x * 2` is executed. The same operations are then executed with the value `1.5`. Finally, the program generates a new DataFrame `df1` with two rows that include `1.0` and `3.0`. When `i = 1`, the value `3.0` is shown.

Listing 5: Example program using the Dataset API

```
1 // ds is stored in column-oriented storage
2 val ds: Dataset[Double] = Seq(0.5, 1.5).toDS.cache
3 for (i <- 0 to 1) {
4   val ds1 = ds.filter(x => x > i).map(x => x * 2)
5   ds1.show
6 }
7 // output  i = 0: 1.0, 3.0
8 //         i = 1: 3.0
```

Listing 6: Logical and physical plans for Listing 5

```
== Logical Plan ==
SerFromObj [input[0, double] AS value]
   +- MapElements <function1>, obj: double
      +- Filter <function1>.apply
         +- DeserToObj value: double, obj: double
            +- InMemRelation [value]
== Physical Plan ==
*SerFromObj [input[0, double] AS value]
   +- *MapElements <function1>, obj: double
      +- *Filter <function1>.apply
         +- *DeserToObj value: double, obj: double
            +- InMemTblScan [value]
               +- InMemRelation [value]
```

Listing 3 shows a pair of logical and physical plans for the program in Listing 2. The physical plan has an `InMemTblScan` operation that reads data from the in-memory cache, which is represented by an `InMemRelation` operation. *Whole-stage code generation* is applied to operations with a `*` prefix in the physical plan. Listing 4 shows the resulting pseudo Java code generated by Catalyst from the physical plan. This code corresponds to the `filter()` and `selectExpr()` operations at line 4 in Listing 2 for `i = 0`. In Listing 4, line 6 uses an iterator to process each row. Line 8 obtains data for a row from `df`. Line 11 obtains a value for column `x`. As the variable `x` is a double primitive type, there is no explicit data-conversion. Lines 13 and 14 execute the `filter()` and `selectExpr()` operations, respectively. Finally, line 17 stores the value for column `x` to a new row, and then line 19 adds the new row to a new DataFrame `df1`.

2.2.2 Dataset API. A Dataset is another data abstraction, which enables programmers to use a lambda expression to be invoked from the Dataset API operations such as `map()` or `filter()`. A Dataset introduces compile-time type safety and more expressive power (e.g., to execute a loop in `map()`) than a DataFrame has. A Dataset can also leverage optimizations used for a DataFrame and use the Tungsten's representation.

Listing 5 shows an example of a Spark program using the Dataset API. This program works the same as that in Listing 2. Listing 6 shows the corresponding pair of logical and physical plans. This logical plan has two additional operations `DeserToObj` and `SerFromObj`, unlike the logical plan in Listing 3. The `DeserToObj` operation converts data from the Tungsten's representation to the Java representation, whereas the `SerFromObj` operation converts data from the Java representation to the Tungsten's representation. Listing 7 shows the generated pseudo Java code for the `filter()` and `map()` operations at line 4 in Listing 5 for `i = 0`. Listing 7 does not show the lambda expressions, which are launched by the `apply()` methods at lines 15 and 18.

Listing 7: Pseudo Java code generated from Listing 5

```
 1 final class GeneratedIterator {
 2   Iterator inputIterator = ...;
 3   Row projectRow = new Row(1);
 4   RowWriter rowWriter = new RowWriter(projectRow);
 5   protected void processNext() {
 6     while (inputIterator.hasNext()) {
 7       // 1. Read a row
 8       Row inputRow = (Row) inputIterator.next();
 9       // 2. Read an element
10       // 3. Convert to Java data format
11       double x = inputRow.getDouble(0)
12       // 4. Perform computation
13       //   apply() calls {x => x > i} for filter
14       //   i is a bounded variable
15       boolean filter_val = (Boolean)filter_func.apply(x);
16       if (!filter_val) continue;
17       //   apply() calls {x => x * 2} for map
18       double map_val = (Double)map_func.apply(x);
19       // 5. Convert to Spark data format
20       // 6. Write an element of each column
21       rowWriter.write(0, map_val);
22       // 7. Write a row
23       appendRow(projectRow);
24     }
25   }
26 }
```

While the physical plan has the `DeserToObj` and `SerFromObj` operations, there is no explicit conversion in Listing 7 owing to the use of a primitive type. Lines 11 and 18 access a primitive double value for column `x`.

2.3 Column-oriented storage

Column-oriented storage is known as an efficient format for relational data processing or data caching [13]. For relational data processing, column-oriented storage enables vectorization. Spark internally stores data in column-oriented storage to read multiple elements at once from a Parquet [4] file. For data caching, column-oriented storage enables a good compression ratio. Spark keeps data materialized by a `cache` operation in column-oriented storage. The DataFrame `df` in Listing 2 and Dataset `ds` in Listing 5 also use column-oriented storage.

3 ANALYSIS AND OPTIMIZATION FOR COLUMN-ORIENTED STORAGE

This section describes problems in code generated from a Spark program to access column-oriented storage. It then describes optimizations to avoid these problems.

3.1 Problems

Column-oriented storage is an important data format used in Spark for performance and memory-reduction purposes. Generated code in Spark, however, always reads data from row-oriented storage or stores data to row-oriented storage even when the code accesses column-oriented storage. This is because the generated code uses an iterator to read data in a row as shown at line 8 in Listings 4 and 7. This behavior involves data-copy or data-conversion between column-oriented and row-oriented storage. These operations degrade performance.

Listing 8: Example program that accesses data in column-oriented storage

```
1 // df and df2 use column-oriented storage
2 val df = Seq(0.5, 1.5).toDF("x").cache
3 val df1 = df.filter("x > 0").selectExpr("x * 2 AS v")
4 val df2 = df1.cache
```

Listing 8 shows an example program that reads data from a cache and stores the data into a new cache in column-oriented storage. DataFrames `df` and `df2` in Listing 8 are stored in column-oriented storage. The generated code for line 2 in Listing 8 is the same as that in Listing 4. The current code-generation phase always generates Java code that reads data from a row by using an iterator and stores data to a row. Line 8 in Listing 4 reads data from row-oriented storage via a `Row` object. The `inputIterator.next()` method is used to convert data from column-oriented to row-oriented storage, as shown in Figure 3. Line 19 then stores the data to row-oriented storage. When a new cache of `df2` is built later, data-copy from row-oriented to column-oriented storage occurs.

3.2 Optimization

Given the above problems, an optimization was devised to eliminate data-copy. This optimization analyzes the data sources and sinks though the following steps:

(1) Check whether a data source for a given operation uses column-oriented storage. If not, go to step 5.
(2) Check whether the data types in each column are supported. The code-generation phase may not support direct read for complicated data types such as `Calender`. If not, go to step 5.
(3) Check whether all of the operations from a data source to a data sink support column-oriented storage. If not, go to step 5.
(4) Generate Java code that directly reads data from column-oriented storage.
(5) If the analysis can identify that a data sink for a given operation uses column-oriented storage and that the data types in each column are supported, then the code-generation phase generates Java code that directly stores data to column-oriented storage.

In step 3, element-wise operations, such as `filter()`, `map()`, and `aggregation()`, can support column-oriented storage without any changes to the operation from the row-oriented storage support. Some complicated operations, such as `join()`, may require additional work to support column-oriented storage. In step 4, because each element of column-oriented storage is accessed to read or write data with a row index, getter and setter methods are necessary. Examples include `getDouble(i)` and `putDouble(i, value)`, where `i` is a row index and `value` is a `double` primitive value to be stored. The number of rows should be known by using a compile-time constant or a method such as `nRows()` before accessing all of the row elements.

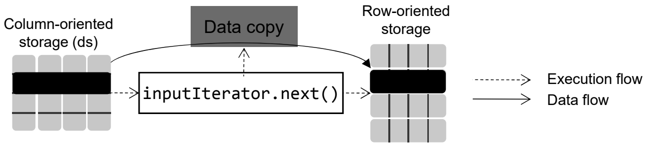


Figure 3: Data copy from column-oriented to row-oriented storage at line 9 in Listing 4

Listing 9: Physical plan for Listing 8

```
== Physical Plan ==
InMemTblScan [value]
   +- InMemRelation [value]
      +- *Project [(x * 2.0) AS v]
         +- *Filter (x > 0.0)
            +- InMemTblScan [x]
               +- InMemRelation [x]
```

Listing 10: Pseudo Java code generated from Listing 8 with the proposed optimization

```
1 final class GeneratedIterator {
2   Columnar columnarSource = ..., columnarSink = ...;
3   ColumnVector col0Source = columnarSource.column(0);
4   ColumnVector col0Sink = columnarSink.column(0);
5   protected void processNext() {
6     for (int i = 0, j = 0; i < columnarSource.nRows(); i++) {
7       // (1) and (2) Read an element from a column
8       // (3) Convert to Java data format
9       //      from column-oriented storage
10      double x = col0Source.getDouble(i);
11      // (4) Perform computation
12      if (!(x > 0)) continue;          // filter("x > 0")
13      double value = x * 2;            // selectExpr("x * 2")
14      // (5) Convert to Spark data format
15      // (6) and (7) Write an element to column-oriented storage
16      col0Sink.putDouble(j++, value);
17    }
18  }
19 }
```

Implementation in Spark: The implementation of this optimization first analyzes a physical plan for a Spark program. Listing 9 shows a physical plan for the program in Listing 8. For lines 2 and 3 in Listing 8, step 1 checks whether the predecessor of the `Filter` and `selectExpr` operations in Listing 8 uses column-oriented storage. The `InMemTblScan` operation in Listing 9 does use column-oriented storage, which corresponds to the `cache` operation. Step 2 checks the data type of each column in the column-oriented storage. In this case, the type is known as `double`, a supported type, based on the type information of the variable `x`. Step 3 checks whether the `Filter`, `selectExpr`, and `Project` operations support column-oriented storage. It is worth noting that the smaller of two inputs for *join* can also support column-oriented storage. Step 5 checks whether the successor of the `Project` operation uses column-oriented storage, which is indeed the case for the `InMemTblScan` operation. Finally, the analysis can prove that the generated code can directly access the column-oriented storage for the data source and sink.

From this analysis result, the code-generation phase generates Java code. Listing 10 shows the pseudo Java code generated from Listing 8. Line 6 introduces a for-loop to iterate all of the rows in column-oriented storage. Then, the generated code directly reads data from column-oriented storage at line 10 and directly writes data to column-oriented storage at line 16 for the `Project` operation. At lines 12 and 13, the `filter()` and `selectExpr()` operations are performed with each row element read at line 10. The loop style is changed from an iterator-based loop in Listing 5 to a simple for-loop here. This change can encourage application of advanced loop optimizations such as loop unrolling in a Java JIT compiler.

4 ANALYSIS AND OPTIMIZATION FOR PRIMITIVE-TYPE ARRAY

This section describes performance problems in generated Java code that accesses a primitive-type array in a Spark program. It then describes optimizations to avoid these problems.

4.1 Problems

Code generation is more complicated for an array than for a scalar value. This is because a variable in a program may have a `null` value (three-valued logic) even for a non-reference variable. Because Java offers only two-valued logic for a non-reference variable, generated Java code has to manage three-valued logic for non-reference elements. In particular, a primitive-type array should be carefully handled to ensure high performance.

Inspection of the Java code generated from a program that uses the DataFrame or Dataset API and accesses a primitive-type array identified the following performance problems:

(1) Data-conversion in an internal array representation class for a primitive-type array.
(2) Data-conversion in generated code .

When the program using the Dataset API in Listing 1 is run, the logical and physical plans in Listing 11 are generated. Then, from this physical plan, the Java code in Listing 12 is generated. To simplify the explanation, while the rest of this section uses a program using the Dataset API as an example, a program using the DataFrame API causes the same problems.

For problem 1 above, the `SerFromObj` operation in the physical plan instantiates a `GenericArrayData` class for a `double` primitive-type array, which corresponds to line 30 in Listing 12. The `GenericArrayData` class can support three-valued logic for an array by using a boxed object. Listing 13 shows a part of this implementation. We found that the object creation in this constructor causes boxing from a `double` value to a `Double` object. This degrades performance.

Listing 11: Logical and physical plans for Listing 1

```
== Logical Plan ==
SerFromObj [new(GenericArrayData) AS value]
   +- MapElements <function1>, obj: [D
      +- DeserToObj cast(value as array<double, containsNull=true>).
      toDoubleArray
         +- Filter <function1>.apply
            +- InMemRelation [value]
== Physical Plan ==
*SerFromObj [new(GenericArrayData) AS value]
   +- *MapElements <function1>, obj: [D
      +- *DeserToObj cast(value as array<double, containsNull=true>).
      toDoubleArray
         +- *Filter <function1>.apply
            +- InMemTblScan [value]
               +- InMemRelation [value]
```

For problem 2, the `DeserToObj` operation in the physical plan executes the type casting from a given array to another array by using a `GenericArrayData` class. The array's data type is then cast to a `double` array by calling a `toDoubleArray()` method, which corresponds to the Java statements at lines 21 to 26 in Listing 12. Similar code is generated for the pair consisting of `InMemTblScan` and `Filter` operations. The `InMemTblScan` operation corresponds to Java statements at lines 11 to 16 in Listing 12. We found that these statements cause boxing from a `double` value to a `Double` object and unboxing from the `Double` object back to a `double` value. As boxing involves an object allocation and unboxing involves an object reference, performance degradation occurs.

4.2 Optimizations

The following optimizations were devised to alleviate the two performance problems above:

(1) Make the internal data representation more efficient to potentially avoid boxing and unboxing at runtime.
(2) Eliminate boxing and unboxing by exploiting static information at compile time.

Regarding optimization 1, a basic idea is to hold a Java data representation in a framework's data structure as much as possible. When the data representation requires a three-valued logic value, however, it is not possible to natively represent that with a Java non-reference variable, which cannot hold `null`. A straight-forward approach is to use a Java object that can hold `null`. While this approach was used in previous versions of Spark for its ease of implementation, it requires boxing and unboxing at runtime when a value is referenced. This degrades performance and consumes additional memory. Our optimization instead keeps the body of a Java primitive-type array with a bitvector to represent the nullability of each element of the array. This can improve performance and decrease memory consumption by avoiding a Java Object array. For this optimization, the internal representation requires getter and setter methods. Examples include `toArray()` that returns a Java array, `fromArray(a)` that accepts a Java array, and `isNull(i)` that returns whether an element is `null`, where `a` is a Java array and `i` is the index of an array element.

Regarding optimization 2, a basic idea is to apply static analysis based on schema information to eliminate unnecessary boxing and unboxing. When the generated code uses a framework's data representation, a straight-forward approach is to always convert an array in the data representation to a Java Object array. This avoids handling the variable type in the generated code. While this approach was used in previous versions of Spark for its ease of implementation, it requires boxing and unboxing at runtime when a primitive-type value is referenced, which again degrades performance and consumes additional memory. Our optimization avoids this data-conversion to a Java Object array by analyzing schema information. Listing 14 is an example of a scheme of

Listing 12: Pseudo Java code generated from Listing 1

```
final class GeneratedIterator {
  Iterator inputIterator = ...;
  Row projectRow = new Row(1);
  RowWriter rowWriter = new RowWriter(projectRow);
  protected void processNext() {
    while (inputIterator.hasNext()) {
      // (1) Read a row
      Row inputRow = (Row) inputIterator.next();
      // (2) Read an array
      ArrayData a = inputRow.getArray(0);
      Object[] obj0 = new Object[a.length];                  /* -----v----- Problem 2 -----v-----*/
      for (int i = 0; i < a.length; i++)
        obj0[i] = Double.valueOf(a.getDouble(i));            /* boxing from double to Double */
      ArrayData array_filter = new GenericArrayData(obj0);   /* -----^----- Problem 2 -----^-----*/
      // (3) Convert to Java data format
      double[] input_filter = array_filter.toDoubleArray();  /* unboxing from Double to double */
      // (4) Perform computation
      boolean fvalue = (Boolean)filter_func.apply(input_filter); // apply() calls {a => a(0) > i} for filter
      if (!fvalue) continue;                                 // i is a bounded variable
      // (2) Read an element
      Object[] obj1 = new Object[a.length];                  /* -----v----- Problem 2 -----v-----*/
      for (int i = 0; i < a.length; i++)
        obj1[i] = Double.valueOf(a.getDouble(i));            /* boxing from double to Double */
      ArrayData array_map = new GenericArrayData(obj1);      /* -----^----- Problem 2 -----^-----*/
      // (3) Convert to Java data format
      double[] input_map = array_map.toDoubleArray();        /* unboxing from Double to double */
      // (4) Perform computation
      double[] mvalue = (double[])map_func.apply(input_map); // apply() calls {a => a} for map
      // (5) Convert to Spark data forma
      ArrayData value = new GenericArrayData(mvalue);        /* Problem 1: boxing from double to Double */
      // (6) Write an array
      rowWriter.write(0, value);
      // (7) Write a row
      appendRow(projectRow);
    }
  }
}
```

Listing 13: Part of the current `GenericArrayData.scala`

```
class GenericArrayData(ary: Array[Any]) // Object[] in Java
    extends ArrayData {
  def this(seq: Seq[Any]) = this(seq.toArray)
  // omitted constructors for other primitive-type arrays
  def this(primAry: Array[Double]) =
    this(primAry.toSeq)     // boxing from double to Double
  ...
  override def getDouble(i: Int): Double =
    ary(i).asInstanceOf[T]  // unboxing from Double to double
  override def toDoubleArray(): Array[Double] = {
    val size = numElements()
    val values = new Array[Double](size)
    for (i <- 0 until size) { values(i) = getDouble(i) }
    values
  }
}
```

Listing 14: Example of schema information for the DataFrame/Dataset API with a primitive-type array

```
val df = Seq(Array(0.5, 0.6),Array(1.5, 1.6)).toDF("a").cache
val ds = Seq(Array(0.5, 0.6),Array(1.5, 1.6)).toDS.cache
// schema for df and ds
root
|-- a: array (nullable = true)
|    |-- element: double (containsNull = false)
```

Listing 15: `GenericArrayData.scala` class enhanced for a primitive-type array

```
class GenericArrayData(val array: Array[Any],
    boArray: Array[Boolean], byArray: Array[Byte],
    sArray: Array[Short], iArray: Array[Int],
    lArray: Array[Long], fArray: Array[Float],
    dArray: Array[Double]) extends ArrayData {
  def this(primAry: Array[Double]) =
   this(null, null, null, null, null, null, null, primAry)
  ...
  override def getDouble(i: Int): Double = {
    if (dArray != null) dArray(i) else array(i).asInstanceOf[T]
  }
  override def toDoubleArray(): Array[Double] = {
    if (dArray != null) dArray else super.toDoubleArray
  }
}
```

a DataFrame or Dataset with a double primitive-type array. If the analysis can prove that there is no *null* element in a given array, then the optimization passes through the array in the data representation without data-conversion. For this optimization, the internal representation requires a setter method, such as, `fromDoubleArray(a)` for the `double` type, where `a` is an array in the data representation.

Implementation in Spark: For optimization 1, we implemented a new `GenericArrayData` class to access array elements without boxing and unboxing. This approach only adds new fields to keep primitive-type arrays, modified source code for constructors (i.e., a setter in our algorithm), and getter methods. Constructors are updated to store a given primitive-type array to the new field. Getter methods such as `getDouble()` and `toDoubleArray()`, are updated to return a value from the new field. There may be a concern regarding space inefficiency in that the new `GenericArrayData` class includes seven additional fields and uses at most one at each time. While the current `GenericArrayData` class keeps an array with a boxed object, however, the new `GenericArrayData` class keeps a primitive-type array without any boxed objects. The total memory space for the new `GenericArrayData` class is thus smaller in practice. This implementation supports only a Java primitive-type array, because our experiences suggests that few primitive arrays have `null` elements. This simplifies the implementation without being concerned with the bit vector. Listing 15 shows part of the enhanced `GenericArrayData` class.

For optimization 2, we implemented specialization of logical plans for the case when the optimizer can recognize that an array element has a primitive-type. The specialization does not generate a type-casting operation to handle three-valued

Listing 16: Pseudo Java code generated from Listing 1 with the two proposed optimizations for a primitive-type array

```
 1 final class GeneratedIterator {
 2   Iterator inputIterator = ...;
 3   Row projectRow = new Row(1);
 4   RowWriter rowWriter = new RowWriter(projectRow);
 5   protected void processNext() {
 6     while (inputIterator.hasNext()) {
 7       // (1) and (2) Read an array from a column
 8       Row inputRow = (Row) inputIterator.next();
 9       ArrayData array = inputRow.getArray(0);
10       // (3) Convert to Java data format
11       double[] input = array.toDoubleArray();
12       // (4) Perform computation
13       boolean filter_va = (Boolean)filter_func.apply(input);
14       if (!filter_val) continue;
15       // (4) Perform computation
16       double[] map_output = (double[])map_func.apply(input);
17       // (5) Convert to Spark data format
18       ArrayData output = new GenericArrayData(map_output);
19       // (6) Write an array
20       rowWriter.write(0, output);
21       // (7) Write a row
22       appendRow(projectRow);
23     }
24   }
25 }
```

logic in Java code but simply allocates a Java primitive-type array from the `GenericArrayData` class instance.

Finally, the proposed optimizations generate the code in Listing 16. As compared with the code in Listing 12, this code is simple, without boxing, unboxing, or type casting. Line 9 still has a data-copy operation using `toDoubleArray()` for a primitive-type array. A previous work [45], however, describes eliminating this operation by using Java bytecode to rewrite the lambda expressions in the `filter` and `map` operations and directly access `ArrayData`, which complement the approach here.

5 EVALUATION

This section presents the results of experimental evaluations of the proposed optimizations on a cluster of five Intel machines running the Ubuntu 16.04 operating system. Each machine has a 16-core Intel Xeon E5-2683 v4 CPU (2.1 GHz with 128 GB of RAM). OpenJDK 1.8.0_181 was used with a 96 GB heap and the default garbage collection policy.

We implemented our optimizations described in Sections 3.2 and 4.2 in Spark 2.2, because they have been integrated into Spark 2.3. Benchmark programs were run with two implementations of Spark: one disabling the optimizations for column-oriented storage and primitive arrays, and the other enabling them. The average results were compared. Each benchmark program was executed 25 times in a single JVM invocation, and the average time of the last 10 executions was reported as a *steady-state* execution time to reduce the impact of JIT compilation anomalies, garbage collection, and other JVM components.

In the cluster of five machines, one was used as a driver, and the other four were used as executors. Each machine launched one executor JVM with 16 worker threads per JVM.

5.1 TPC-H queries

This section discusses the performance improvements for TPC-H[42] queries to a relational database system. All of the 22 queries in the TPC-H benchmark were considered with a scale factor of 10. The data was stored to an in-memory cache, and a query was executed five times while reading data from the cache. The effectiveness was measured only in terms of optimizations for column-oriented storage, because the schema of TPC-H includes only the scalar data types `integer`, `double`, and `String`, but no array type. It is worth noting that the optimization was enhanced for the `String` type.

The performance was measured for all of the queries on a cluster. Specifically, the elapsed time was measured from start to completion of executing each query. Figure 4 shows thee performance improvement with the optimization to eliminate data-copy for column-oriented storage. The Y-axis represents the ratio of elapsed times. The optimization improved the performance of the 22 queries by a geometric mean of 1.10×. The performance of six queries improved by more than 1.10×, and in particular, the performance of query 6 (Q6) improved by 1.41×. This was because the processing time for reading tables is longer for Q6 than for the other queries. This query includes only lightweight operations, such as `filter` and `aggregation`, and does not have heavier operations, such as `sort` and `join`. In contrast, Q1, Q11, Q12, Q15, and Q22 have a cached table followed by a sequence of `filter` or aggregation operations, and other operations such as `sort` and `join`. As the current implementation does not support `sort` or some `join` operations designed for row-oriented storage, it is not effective for such cases of reading a cached table followed by these operations. The proposed optimization is clearly effective, however, for some types of queries in a database system.

5.2 Machine-learning algorithms

This section shows the performance improvement for two machine-learning algorithms:

- K-means: iterative-clustering algorithm
- Logistic regression: iterative-classification algorithm with a regression model

These benchmark programs, originally used RDDs [19] and an example in the Spark package, but we rewrote them for this work using the Dataset API. This was because versions of these two algorithms using the Dataset API cannot be found in current Spark benchmark suites such as SparkBench [27] and Spark Performance Tests [14]. Data read more than once was stored to an in-memory cache. A 5M-point data set with 200 dimensions and a 32M-point data set with 200 dimensions were used for k-means and logistic regression, respectively. The elapsed time was measured from start to completion of the computation in each program. Figure 5 shows the performance improvement with the proposed optimizations. The Y-axis shows the relative performance with respect to no optimization. For each algorithm, the left bar in red shows the performance with the two optimizations

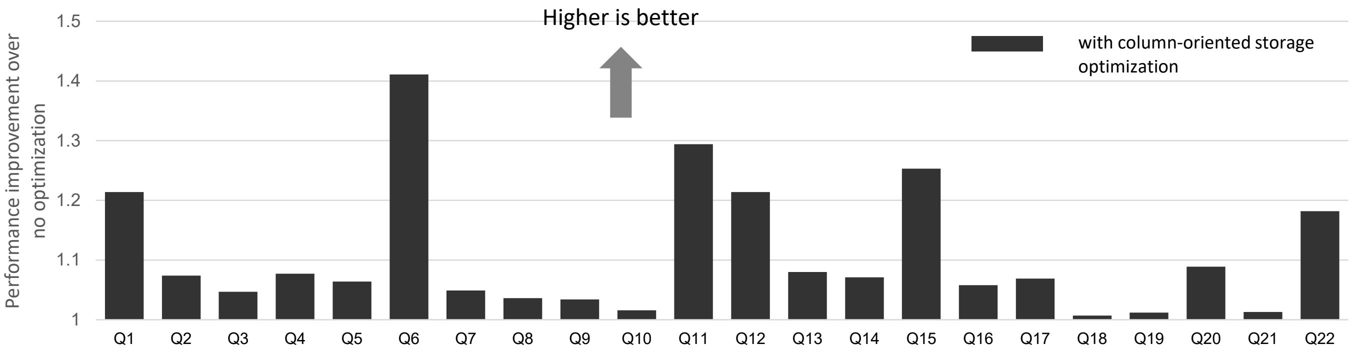


Figure 4: Performance comparison for TPC-H queries with and without the optimization for column-oriented storage on cluster

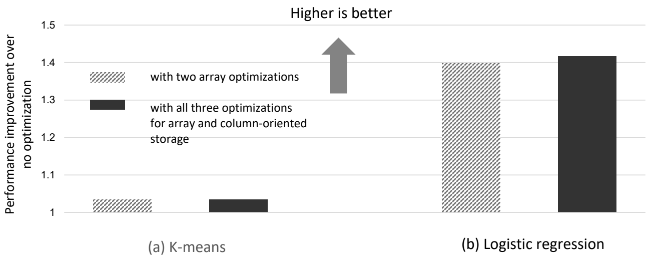


Figure 5: Performance comparison for machine-learning algorithms with and without the optimizations for dealing with primitive-type arrays and column-oriented storage on cluster

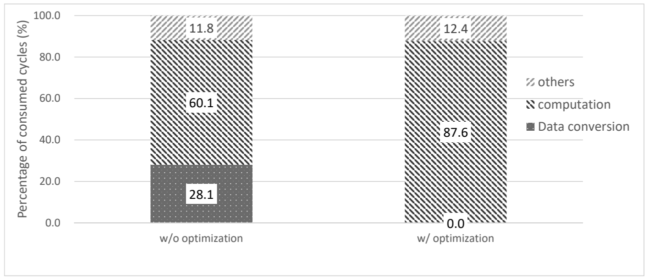


Figure 6: Method-level profiling for logistic regression with and without the primitive-type array optimizations

for eliminating data-conversion in accessing a primitive-type array. The right bar in blue shows the performance with all of three optimizations, including the one for eliminating data-copy in accessing column-oriented storage. The optimizations improved the performance for k-means by 1.02× and logistic regression by 1.42× (for a geometric mean of 1.21×) by eliminating data-conversion for accessing a primitive-type array. The proposed optimizations were thus also effective for machine-learning workloads.

To further investigate why the optimizations in eliminating data-conversion for accessing a primitive-type array, method-level profiling was applied using the `perf` command. Figure 6 shows the results for the logistic regression algorithm with and without the optimizations. The graph presents a breakdown of all CPU cycles in the JVM. The left bar shows the breakdown without the optimizations, while the right bar shows the breakdown with them. The optimizations eliminated cycle consumption for executing data-copy with object boxing. The proportion of computation improved from 60.1 to 87.6%. This graph shows that the overall performance improved because of a reduction in cycle consumption for the data-conversion. From this profiling, cycle consumption for accessing column-oriented storage was not dominant, as computation accounted for most of the cycles. As a result, the optimization for eliminating data-copy in accessing column-oriented storage resulted in little performance improvement for the logistic regression algorithm.

6 RELATED WORK

This section describes related work in several areas: Apache Spark, compilation and optimizations of DSLs, column-oriented storage format, and data-conversion between a custom data representation and Java-data representation.

6.1 Apache Spark

Apache Spark [48] is an in-memory computing framework. It executes computations to transform an RDD [47] to a new RDD by parallel operations such as `map()`, `filter()`, or `reduce()`. Spark provides several domain specific libraries for applications such as machine learning [31], graphs [17], and the R programming language [43]. Spark 2.0 and later versions generates Java code from a Spark program to achieve high performance [2, 6].

There have been many studies on improving the performance of Apache Spark workloads. Chiba and Onodera [12] improved the performance of TPC-H queries by 1.4× by tuning option parameter values for a JVM. Jia et al. [24] improved the performance of machine-learning applications by 1.6× by tuning the available number of simultaneous threads for multithreading on a core. Taneja et al. [41] improved the performance of a machine-learning application by 2.2× by tuning option parameter values for Spark. Chaimov et al. [10] identified bottlenecks for a shuffle operation and in-memory cache with Lustre and improved the performance by adding file pooling and a large non-volatile RAM (NVRAM) buffer. Ousterhout et al. [33] identified many stragglers in Spark through block time analysis and revealed undiscovered opportunities for performance improvements of TPC-DS queries. Lu et al. [28] improved the performance of the sort benchmark by 1.8× by exploiting remote direct memory access (RDMA) over InfiniBand.

While there have been many studies like those above, few have been conducted to analyze generated code in Spark. Canali [9] analyzed generated code through sampling profiling. Essertel et al. [16] showed performance issues at access and decoding with in-memory data representations, also through sampling profiling. Grossman and Sarkar [19] devised a framework to generate GPU code from a Spark program using the RDD API. That work also involved data-conversions between row-oriented and column-oriented storage formats. Wroblewski et al. [45] improved the performance of a Spark workload, that accesses a primitive-type array by rewriting the Java bytecode of a lambda expression. They reported that a program using the Dataset API performs better than a program using the RDD API. To the best of our knowledge, however this paper is the first to analyze generated code by inspecting the code and to optimize it by devising optimizations in Spark. The optimizations here an be applied complementarily with those of previous studies.

6.2 Compilation of DSLs

Multiple data processing frameworks generate native code from a given program. Impala [25] translates a SQL query into native code by using a low-level virtual machine (LLVM) compiler [26] with volcano style [44]. That paper does not, however, describe the details of code generation for column-oriented storage or a primitive-type array. Flare [16] uses a very promising approach that generates machine code from a Spark program by using a completely different advanced toolchain, *Delite*, which was designed for a DSL [40], instead of using Catalyst. Flare achieves high performance in processing TPC-H queries. The present work complementarily addresses performance bottlenecks in Spark versions before 2.3. Weld [34] is a new approach for defining a common intermediate representation to optimize across different frameworks. It uses the an LLVM compiler as a backend instead of a JIT compiler in a JVM. As the current version of Weld does not have a `null` value and does not use column-oriented storage, it does not have the related performance problems.

Deep learning frameworks generate native code from a given program using an embedded DSL. TensorFlow XLA [1], PyTorch [35], and TVM [11] translate their programs into native code for each target device by using the LLVM compiler.

An especially important DSL is SQL. In SQL, volcano style [18] is a basic execution model for a query to pass data by using an iterator between two operations. Volcano style has the efficiency advantages of pipeline processing. MonetDB [8] uses an extended volcano style to support column-oriented storage through buffering.

Neumann [32] described an approach for generating native code from an SQL query by using the LLVM compiler. To leverage the latest compiler optimizations, this approach eliminates iterators for Volcano style by putting multiple operations into one function. Catalyst also uses this approach. Our optimizations were created on top of this research.

Rao et al. [37] described an approach to dynamically generate Java code for each query. The generated code improves the query performance by using a JIT compiler in JVM. The code generation phase in Catalyst uses that idea. Rao et al. used volcano style to easily switch the query execution engine for each operation between an interpreter or generated code. They did not, however, describe optimizations at the code generation phase.

6.3 Column-oriented storage format

Column-oriented storage [13] is a well-known format used by many database systems [8, 36] and Spark. Daniel et al. [29] described a strategy of selectively using row- or column-oriented storage. The code generation optimization there were complementarily devised for column-oriented storage. DB2 is a state-of-the-art production system uses both column- and row-oriented storage and leverages SIMD instructions to exploit instruction-level parallelism. Exploitation of SIMD in collaboration with a JIT compiler remains as a future work.

6.4 Data-conversion

A JVM is the foundation of various runtime frameworks because of its platform-neutral nature. Because a JVM has some memory management overhead for a managed runtime environment, some runtime frameworks manage memory allocations and accesses through a customized approach using the Java Unsafe API [30]. While that approach can result in high performance within a system, it introduces a new challenge in the data-conversion overhead between a custom data representation and the Java-data representation for a bridge with a method written in a JVM language. The optimizations here alleviate this overhead by exploiting schema information.

7 CONCLUSION

The performance bottlenecks of Apache Spark, which is a widely used open-source framework, were analyzed by inspecting generated code. This inspection identified two performance issues: data-copy and data-conversion. We devised optimizations to eliminate these two performance issues, and then implemented them for Spark. The performance for benchmark programs improved by up to 1.4× on a cluster. This code inspection and optimization approach can be applied to other big data framework to improve performance.

ACKNOWLEDGMENTS

We thank Wenchen Fan, Herman van Hovell, Liang-Chi Hsieh, Tayuka Ueshin, Sameer Agarwal, Andrew Or, Davies Liu, Nong Li, and Reynold Xin for their suggestions on merging our optimizations into Apache Spark.

REFERENCES

[1] Martín Abadi, Michael Isard, and Derek G. Murray. 2017. A Computational Model for TensorFlow: An Introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2017)*. 1–7.

[2] Sameer Agarwal, Davies Liu, and Reynold Xin. 2016. Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop. (2016). https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html

[3] Apache Hadoop. 2007. (2007). https://hadoop.apache.org.

[4] Apache Parquet. 2013. (2013). https://parquet.apache.org.

[5] Michael Armbrust, Wenchen Fan, Reynold Xin, and Matei Zaharia. 2016. Introducing Apache Spark Datasets. (2016). https://databricks.com/blog/2016/01/04/introducing-apache-spark-datasets.html

[6] Michael Armbrust, Reynold 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 *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. 1383–1394.

[7] Ahsan Javed Awan, Mats Brorsson, Vladimir Vlassov, and Eduard Ayguade. 2015. Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server. In *Proceedings of the 2015 IEEE Fifth International Conference on Big Data and Cloud Computing (BDCLOUD '15)*. 1–8.

[8] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Conference on Innovative Data Systems Research (CIDR)*. 225–237.

[9] Luca Canali. 2016. Voice from CERN: Apache Spark 2.0 Performance Improvements Investigated With Flame Graphs. (2016). https://databricks.com/blog/2016/10/03/voice-from-cern-apache-spark-2-0-performance-improvements-investigated-with-flame-graphs.html

[10] Nicholas Chaimov, Allen Malony, Shane Canon, Costin Iancu, Khaled Z. Ibrahim, and Jay Srinivasan. 2016. Scaling Spark on HPC Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*. 97–110.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[12] Tatsuhiro Chiba and Tamiya Onodera. 2016. Workload characterization and optimization of TPC-H queries on Apache Spark. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 112–121.

[13] George P. Copeland and Setrag N. Khoshafian. 1985. A Decomposition Storage Model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data (SIGMOD '85)*. 268–279.

[14] Databricks. 2014. Spark Performance Tests. (2014). https://github.com/databricks/spark-perf/

[15] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI'04)*. 10–10.

[16] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, and Tiark Rompf. 2018. Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 799–815.

[17] Joseph E. Gonzalez, Reynold Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 599–613.

[18] Goetz Graefe. 1994. Volcano — An Extensible and Parallel Query Evaluation System. *IEEE Transaction on Knowledge and Data Engineering* 6, 1 (1994), 120–135.

[19] Max Grossman and Vivek Sarkar. 2016. SWAT: A Programmable, In-Memory, Distributed, High-Performance Computing Platform. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC '16)*. 81–92.

[20] Kazuaki Ishizaki. 2016. SPARK-15985: Eliminate redundant cast from an array without null or a map without null. (2016). https://github.com/apache/spark/pull/13704

[21] Kazuaki Ishizaki. 2016. SPARK-16213: Reduce runtime overhead of a program that creates an primitive array in DataFrame. (2016). https://github.com/apache/spark/pull/13704

[22] Kazuaki Ishizaki. 2016. SPARK-17490: Optimize SerializeFromObject() for a primitive array. (2016). https://github.com/apache/spark/pull/13704

[23] Kazuaki Ishizaki. 2017. SPARK-20822: Generate code to directly get value from ColumnVector for table cache. (2017). https://github.com/apache/spark/pull/13704

[24] Zhen Jia, Chao Xue, Guancheng Chen, Jianfeng Zhan, Lixin Zhang, Yonghua Lin, and Peter Hofstee. 2016. Auto-tuning Spark Big Data Workloads on POWER8: Prediction-Based Dynamic SMT Threading. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. 387–400.

[25] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Conference on Innovative Data Systems Research (CIDR)*.

[26] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. 75–86.

[27] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. 2015. SparkBench: A Comprehensive Benchmarking Suite for in Memory Data Analytic Platform Spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers (CF '15)*. 53:1–53:8.

[28] Xiaoyi Lu, Dipti Shankar, Shashank Gugnani, and Dhabaleswar K. Panda. 2016. High-performance design of Apache Spark with RDMA and its benefits on various workloads. In *2016 IEEE International Conference on Big Data, BigData 2016*. 253–262.

[29] Samuel R. Madden, Daniel S. Myers, David J. DeWitt, and Daniel J. Abadi. 2007. Materialization Strategies in a Column-Oriented DBMS. *2007 IEEE 23rd International Conference on Data Engineering* (2007), 466–475.

[30] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. Use at Your Own Risk: The Java Unsafe API in the Wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. 695–710.

[31] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.

[32] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[33] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. 293–307.

[34] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. 2017. Weld: A Common Runtime for High

Performance Data Analytics. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[35] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[36] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proceedings of the VLDB Endowment* 6, 11 (Aug. 2013), 1080–1091.

[37] Jun Rao, Hamid Pirahesh, C. Mohan, and Guy Lohman. 2006. Compiled Query Execution Engine Using JVM. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*. 23–23.

[38] Juwei Shi, Yunjie Qiu, Umar Farooq Minhas, Limei Jiao, Chen Wang, Berthold Reinwald, and Fatma Özcan. 2015. Clash of the Titans: MapReduce vs. Spark for Large Scale Data Analytics. *Proceedings of the VLDB Endowment* 8, 13 (Sept. 2015), 2110–2121.

[39] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. 2012. M3R: Increased Performance for In-memory Hadoop Jobs. *Proceedings of the VLDB Endowment* 5, 12 (Aug. 2012), 1736–1747.

[40] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. 2013. Composition and Reuse with Compiled Domain-specific Languages. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*. 52–78.

[41] Rohit Taneja, Raj Krishnamurhty, and Gang Liu. 2016. In *The 2016 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'16)*.

[42] The Transaction Processing Council. 2017. TPC-H Standard Specification Revision 2.17.3. (2017).

[43] Shivaram Venkataraman, Zongheng Yang, Davies Liu, Eric Liang, Hossein Falaki, Xiangrui Meng, Reynold Xin, Ali Ghodsi, Michael Franklin, Ion Stoica, and Matei Zaharia. 2016. SparkR: Scaling R Programs with Spark. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. 1099–1104.

[44] Skye Wanderman-Milne and Nong Li. 2014. Runtime Code Generation in Cloudera Impala. *IEEE Data Engineering Bulletin* 37, 1 (2014), 31–37.

[45] Jan Wroblewski, Kazuaki Ishizaki, Hiroshi Inoue, and Moriyoshi Ohara. 2017. Accelerating Spark Datasets by inlining deserialization. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2017)*.

[46] Reynold Xin and Josh Rosen. 2015. Project Tungsten: Bringing Apache Spark Closer to Bare Metal. (2015). https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. 1.

[48] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. 1.