Towards Dynamic SQL Compilation in Apache Spark

Filippo Schiavio
filippo.schiavio@usi.ch
Università della Svizzera italiana
Lugano, Switzerland

Daniele Bonetta
daniele.bonetta@oracle.com
Oracle Labs - VM Research Group
Cambridge, MA, USA

Walter Binder
walter.binder@usi.ch
Università della Svizzera italiana
Lugano, Switzerland

ABSTRACT

Big-data systems have gained significant momentum, and Apache Spark is becoming a de-facto standard for modern data analytics. Spark relies on code generation to optimize the execution performance of SQL queries on a variety of data sources. Despite its already efficient runtime, Spark's code generation suffers from significant runtime overheads related to data de-serialization during query execution. Such performance penalty can be significant, especially when applications operate on human-readable data formats such as CSV or JSON.

CCS CONCEPTS

• **Software and its engineering → Source code generation**.

KEYWORDS

Apache Spark SQL, SQL Compilation, Dynamic Compilation

ACM Reference Format:
Filippo Schiavio, Daniele Bonetta, and Walter Binder. 2020. Towards Dynamic SQL Compilation in Apache Spark. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3397537.3397566

1 INTRODUCTION

Big-data systems such as Apache Spark [26] or Apache Flink [3] are becoming de-facto standards for distributed data processing. Their adoption has grown at a steady rate over the past years in domains such as data analytics, stream processing, and machine learning. One of the key advantages of systems such as Apache Spark over their predecessors (e.g., the Hadoop [19] MapReduce framework) is the extensive availability of high-level programming models, coupled with the support for a vast variety of data formats, from plain text files to compressed and highly efficient columnar binary formats such as Apache Parquet [21]. One of the most popular programming models supported by many big-data systems, including Spark, is the SQL programming language.

The Spark SQL API [1] can be conveniently used by data scientists to perform analytical queries over structured or semi-structured data without having to import the data into a database management system (DBMS). The popularity of the SQL language together with the ability to execute queries over raw data (e.g., directly on a CSV file) are two of the key reasons for the increasing popularity of systems like Spark, and represent appealing features for data scientists, who often need to combine analytical workloads (easily expressed in SQL) with numerical computing, for example in the context of large statistical analyses (expressed in Python or R). Another important advantage comes from the opportunity to optimize data processing by applying many results achieved by the relational-database community during the last decades, thanks to the well-studied topic of query-plan optimization [4–6].

In this paper, we describe how SQL compilation is implemented in Apache Spark and some missing optimization opportunities in the code generation. Moreover, we discuss our approach to overcome such inefficiencies and we report our preliminary results.

2 BACKGROUND

When executing an SQL query, Spark generates a query execution plan [17], i.e., an intermediate representation of the query execution at various abstraction layers (e.g., logical and physical, like the ones generated by a traditional DBMS). The physical plan in Spark is a directed acyclic graph (DAG) representing the actual query computation steps (e.g., file scan, filters, projections, joins, sorting, and aggregations) in the distributed cluster. Then, Spark SQL relies on code generation to optimize the DAG: an SQL query is compiled into one or more Java classes, which are responsible for the distributed query execution on a cluster of machines or in cloud deployments. The generation of Java code plays a focal role in Spark's SQL execution pipeline, as it is responsible for ensuring that the entire analytical workload can efficiently execute in a distributed way, using internal Spark abstractions such as Resilient Distributed Datasets [25] (RDDs). In addition to performance, Spark's SQL code generation plays another fundamental role in the Spark ecosystem, as it corresponds to a language-independent API: data scientists and developers can process the results of an SQL query with any language supported by Spark (Java, Scala, R, Python).

Compiling SQL to optimize the execution of a query at runtime has become common in commercial databases (e.g., Oracle RDBMS [14], MonetDB [9], PrestoDB [22], MapDB [20], etc.). However, unlike traditional database systems, Spark SQL compilation does not target a specific data format (e.g., the columnar memory layout used by a specific database system), but typically targets *all* data formats supported by the platform within a single version of the generated code. Due to the great number of such formats, Spark SQL generates highly polymorphic code. In this way, Spark separates data access (i.e., parsing and de-serializing the input data) from the actual data processing: in a first step, data is read from the data source (e.g., a CSV file) and is stored in memory; in a second step, the generated code is executed to perform the actual query execution on the in-memory data. In this way, the same compilation pipeline can be re-used to target multiple data formats, without having to extend the SQL compiler back-end for new data formats.

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal
© 2020 Copyright held by the owner/author(s).
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal*, https://doi.org/10.1145/3397537.3397566.

```
class GeneratedBySpark {
  int dateFrom = dateToInt("2020-03-23");
  int dateTo = dateToInt("2020-03-26");

  public void compute(Data input) {
    while (input.hasNext()) {
      // parse the next CSV input data
      Row row = input.parseNext();
      // string comparison
      UTF8String city = row.getUTF8String("city");
      if(!city.equals("Porto")) continue;
      // date range check
      int date = row.getDate("shipdate");
      if (date < dateFrom) continue;
      if (date > dateTo) continue;
      // accumulate value 'price' as result
      double price = row.getDouble("price");
      accumulate(price);
    }
  }
}
```

Figure 1: Java code produced by Spark for the example query

As an example, consider the following SQL query.

```
SELECT SUM(price)
FROM orders
WHERE city = 'Porto'
    AND shipdate BETWEEN
        date '2020-03-23' AND date '2020-03-26'
```

The Java code that Spark generates at runtime for the example query is depicted in Figure 1[1]. As the figure shows, Spark generates Java code to process the input data file line-by-line. The generated Java code relies on explicit runtime calls to internal Spark components to execute certain data-processing tasks. For example, the generated code calls `parseNext()` to parse the CSV input data, allocating one or more Java object for each input element. All such calls to internal Spark components have the advantage of not requiring to change the code generation depending on the input data format: in fact, the generated code in Figure 1 can execute the query on CSV files, on JSON files, as well as on any other supported data format for which Spark has an implementation of `parseNext()`.

A downside of Spark's modular code generation separating data de-serialization from data processing is performance. Indeed, it is known [11] that the total cost of a query execution on JSON data is dominated (>80%) by JSON parsing, even for complex queries that involve joins and aggregations. Specifically, the code in Figure 1 presents two significant limitations that may impair SQL execution performance:

(1) Eager parsing of the input data: each single row is parsed by a general-purpose de-serializer (e.g., a CSV parser) that consumes the *entire* body of the input data. This is potentially a significant waste of resources, since parsing each single CSV value in a very large file means allocating many temporary, short-lived objects in the JVM heap memory space. Short-lived values are not always needed to execute the entire query: depending on the number of fields needed by the given query, and its selectivity [12], limiting parsing to a subset of the elements may already be enough to filter out values that are not relevant for the query evaluation.
(2) General-purpose predicate evaluation: each predicate involved in the query execution has a *generic* implementation. I.e., the generated code does not take into account the underlying data format, since all the input values are converted into the Spark internal representation during parsing, then predicates are evaluated on such representations.

As we will discuss in the rest of this paper, generality in SQL compilation comes at the price of performance. In contrast, we argue that code generation should be *specialized* as much as possible, taking into account both static and dynamic information about the executed query.

3 BETTER CODE GENERATION IN SPARK

To highlight and overcome some of the missing optimization opportunities of Spark's code generation, we implemented a prototype which jointly optimizes data access to text-based data formats and the predicate evaluation on raw data. Our SQL compiler relies on the intuition that data de-serialization and predicate evaluation should be specialized according to the input data schema and other aspects of an SQL query. Specifically, we believe that the generated code should include a parser specialized for the given query, instead of relying on a general purpose one. Moreover, the generated code should be able to adapt at runtime based on the specific properties of the data. To this end, we are investigating dynamic optimization techniques for predicate evaluation.

3.1 Specialized Data Access

Our SQL compiler generates executable Java code that combines input data parsing with actual SQL execution. Data access (e.g., CSV parsing) is specialized according to the fields used by the query and their declaration order in the input dataset. The CSV parsing approach is based on the intuition that the compiled SQL code should parse only the values that are needed, and should parse them incrementally, that is, lazily rather than eagerly. In particular, the generated parser should be able to skip all the fields which are not used by the given query. Moreover, the order in which fields are read during query evaluation should follow the order in which the data is actually consumed. This information is not used by the Spark SQL code generator, but can be exploited to optimize query execution even further by *avoiding* parsing values that are not needed by the query. By re-ordering the evaluation of query predicates where possible, the parsing operation can be executed in a single step, instead of converting the byte array into a single Java `String` object and then into a `String` array, as Spark currently does. As an example, consider the CSV input data shown in Figure 2. To execute the example query from Section 2, the main loop body of the code generated with our approach consists of (1) skipping the value of the irrelevant field `id`, (2) storing the position of the

[1]Note that this is a simplified version; the `accumulate` operation adds the value `price` to a local accumulator which will be sent as input to the next phase that sums up all local accumulators returned by Spark executors in the cluster.

```
|id:int|price:decimal|shipdate:date|city:string|.....
|1     |9.95         |2020-03-24   |Porto      |.....
```

Figure 2: Example input value and schema for the CSV values in the example

field `price` so that it can be retrieved later (if the predicate passes), (3) evaluating the predicate on field `shipdate`, (4) evaluating the predicate on field `city` and (5) materializing the value of field `price`.

A similar approach can be applied when executing queries on JSON data sets. However, values in JSON are not necessarily always declared in the same order, and it is possible for a given value to appear at different positions in two different JSON objects. In this scenario, access to JSON data can be optimized by the generated code using a speculative approach [2]. Specifically, the JSON parser code can be generated based on the assumption that the input JSON objects will match a given JSON structure; if successful, the parsing operation can be performed with higher performance; if not, a general-purpose JSON parser is used to carry out a full JSON parsing operation.

3.2 Specialized Predicate Execution

As discussed, generating a query-specialized parser integrated with the code that actually executes the query allows skipping unnecessary de-serialization steps. Moreover, integrating a parser within the generated code allows one to specialize the predicate execution too, by generating code which takes into account the underlying data format, rather than relying on general purpose comparisons among heap allocated Java objects.

Consider the code that executes the date range and string equality predicates shown in Figure 1. The code generated by Spark SQL suffers from the following inefficiencies:

(1) Data materialization: once the value for `shipdate` has been parsed from the input data (CSV or JSON), the corresponding value is converted from a sequence of bytes to an heap-allocated Java `String`, then converted into a `Date` object and finally into a primitive `int`. This operation introduces extra conversion overhead, since allocating many objects with a short life-time (i.e., the processing time of a single tuple) may put the JVM's garbage collector under pressure.
(2) Comparisons: once converted to a Java value, the comparison between the input value and the expected constant is performed using the general comparison operator (e.g., `UTF8String.equals`). This operation is efficient as long as the data type is a primitive Java type, but may introduce runtime overhead for Spark data types such as `UTF8String`.

Predicate evaluation can be performed more efficiently by leveraging the static information available at SQL compilation time. In this example, such information include the underlying data format (i.e., `UTF-8` in this case) and the expected operations to be performed (i.e., a date range check and a string equality operations in our example), as well as the expected date format (i.e., `'yyyy-MM-dd'`). Specifically, the compiler can generate machine code that is capable of performing the predicates (1) without allocating new objects nor pushing primitives on the stack, and (2) evaluating each condition (i.e., `'2020-03-23' <= shipdate <= '2020-03-26'` and `city == "Porto"`) on the raw input byte array. Such an optimized comparison can be executed in a single pass in most cases, but may not always be successful (e.g., a date may not match the expected format). Therefore, the predicate evaluation code can be generated under speculative assumptions, so that it falls back to the Spark generic implementation if the speculative assumptions do not hold.

3.3 Results Overview

Our prototype [16] is implemented on top of the Truffle framework [24] and relies on its optimization and de-optimization capabilities to dynamically (re)compile the generated code based on runtime properties of the processed data. Our novel SQL compiler outperforms the state-of-the-art Spark SQL compiler in all TPC-H queries [23], with speedups up to 8.45x for CSV and 4.9x for JSON in a local setting (i.e., executing Spark on a single machine) and up to 4.4x for CSV and 2.6x for JSON in a distributed setting .

4 RELATED WORK

The execution time of SQL queries on text-based data formats, in particular for simple queries, is dominated by data de-serialization time [11]. Thus, different approaches have been proposed to address the parsing problem in the context of SQL query processing. Mison [11] is a recent work which pushes down both predicates and projections to JSON parsing (i.e., to execute both predicates and projections during the parsing phase, instead of doing it later). In particular, the idea behind Mison is to speculate over the value positions (i.e., to avoid a full-scan operation to find the begin and end positions of fields) and to use SIMD instructions to vectorize the scanning operation. Another recent technique that takes advantage of predicate push-down is Sparser [15]. The idea behind Sparser is to execute a pre-filtering phase over raw data which can discard rows that surely do not pass the predicate. Generally, such a pre-filtering phase is not completely accurate, since a row that passes the pre-filtering stage may still be discarded later by the actual filters. Hence, using Sparser, such a pre-filtering phase avoids parsing a subset of data rows when is known that it does not pass a specific predicate.

To the best of our knowledge, no existing work optimizes the parsing phase within Spark code generation and, instead, the existing approaches are usually implemented in external C programs and executed by Spark using the Java Native Interface (JNI) [13], as required by both Mison and Sparser. Since JNI incurs additional invocation costs [7, 8, 10, 18], we believe that a better code generation can overcome the need to invoke statically compiled applications from dynamic libraries to achieve better performance.

5 CONCLUSION

SQL query compilation in Apache Spark relies on static code generation: once compiled to Java code, a query is executed without any further interactions with the query compiler. Static compilation has the main limitation that runtime information cannot be exploited by the generated code. In contrast, we believe that SQL compilation should be dynamic: after an initial compilation of the query, the compiled query code should be able to perform runtime profiling in order to take into account any aspect of a query execution, and

modify the behavior of the query execution by triggering code optimization and de-optimization accordingly.

We are currently working on a prototype which enables dynamic optimizations in the Spark-SQL code generation. Our initial prototype is focused on the so-called "physical operation" of a query, i.e., the leaves of the query plan generated by Spark (i.e., file scans with filter, and projection operations). We are also investigating other optimization opportunities in the context of other SQL operators (e.g., joins and aggregations). Moreover, we are interested in extending our approach to other data formats, e.g., Apache Parquet.

ACKNOWLEDGMENTS

The work presented in this paper has been supported by Oracle (ERO project 1332) and the Swiss National Science Foundation (project 200020_188688). We thank the VM Research Group at Oracle Labs for their support. Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

REFERENCES

[1] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark *(SIGMOD)*. 1383–1394.

[2] Daniele Bonetta and Matthias Brantner. 2017. FAD.js: fast JSON data access using JIT-based speculative optimizations. *Proc. VLDB Endow.* (2017), 1778–1789.

[3] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* (2015), 28–38.

[4] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* (1995), 19–29.

[5] Goetz Graefe and David J. DeWitt. 1987. The EXODUS Optimizer Generator. *SIGMOD Rec.* (1987), 160–172.

[6] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*. 209–218.

[7] Matthias Grimmer, Manuel Rigger, Lukas Stadler, Roland Schatz, and Hanspeter Mössenböck. 2013. An Efficient Native Function Interface for Java. In *PPPJ*. 35–44.

[8] Nassim A. Halli, Henri-Pierre Charles, and Jean-François Méhaut. 2014. Performance comparison between Java and JNI for optimal implementation of computational micro-kernels. *CoRR* (2014).

[9] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin* (2012), 40–45.

[10] Dawid Kurzyniec and Vaidy Sunderam. 2001. Efficient Cooperation between Java and Native Codes - JNI Performance Benchmark. In *PDPTA*.

[11] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *Proc. VLDB Endow.* (2017), 1118–1129.

[12] Clifford A. Lynch. 1988. Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distribution of Column Values. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB '88)*. 240–251.

[13] Oracle. 2019. *Java Native Interface Specification Contents.* https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html

[14] Oracle RDBMS. 2019. *Database | Cloud Database | Oracle.* https://www.oracle.com/it/database/

[15] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter Before You Parse: Faster Analytics on Raw Data with Sparser. *Proc. VLDB Endow.* (2018), 1576–1589.

[16] Filippo Schiavio, Daniele Bonetta, and Walter Binder. 2020. Dynamic Speculative Optimizations for SQL Compilation in Apache Spark. *Proc. VLDB Endow.* (2020), 754–767.

[17] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD*. 23–34.

[18] Levon Stepanian, Angela Demke Brown, Allan Kielstra, Gita Koblents, and Kevin Stoodley. 2005. Inlining Java Native Calls at Runtime. In *VEE*. 121–131.

[19] Team Apache Hadoop. 2019. *Apache Hadoop.* https://hadoop.apache.org/

[20] Team MapDB. 2019. *MapDB.* http://www.mapdb.org/

[21] Team Parquet. 2019. *Apache Parquet.* https://parquet.apache.org/

[22] Team PrestoDB. 2019. *Presto | Distributed SQL Query Engine for Big Data.* http://prestodb.github.io/

[23] Team TPC. 2019. *TPC-H - Homepage.* http://www.tpc.org/tpch/

[24] Christian Wimmer and Thomas Würthinger. 2012. Truffle: A Self-optimizing Runtime System. In *SPLASH*. 13–14.

[25] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*. 15–28.

[26] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud*. 10–10.