Dynamic Speculative Optimizations for SQL Compilation in Apache Spark

Filippo Schiavio
Università della Svizzera
italiana (USI)
Switzerland
filippo.schiavio@usi.ch

Daniele Bonetta
VM Research Group
Oracle Labs
USA
daniele.bonetta@oracle.com

Walter Binder
Università della Svizzera
italiana (USI)
Switzerland
walter.binder@usi.ch

ABSTRACT

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

In this paper we present a new approach to query compilation that overcomes these limitations by relying on runtime profiling and dynamic code generation. Our new SQL compiler for Spark produces highly-efficient machine code, leading to speedups of up to 4.4x on the TPC-H benchmark with textual-form data formats such as CSV or JSON.

PVLDB Reference Format:
Filippo Schiavio, Daniele Bonetta, Walter Binder. Dynamic Speculative Optimizations for SQL Compilation in Apache Spark. *PVLDB*, 13(5): 754-767, 2020.
DOI: https://doi.org/10.14778/3377369.3377382

1. INTRODUCTION

Data processing systems such as Apache Spark [47] or Apache Flink [4] 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. Two key advantages of systems such as Apache Spark over their predecessors (e.g., Hadoop [32]) are the availability of high-level programming models such as SQL, and the support for a great variety of input data formats, ranging from plain text files to very efficient binary formats [39]. The convenience of SQL together with the ability to execute queries over raw data (e.g., directly on a JSON file) represent appealing features for data scientists, who often need to combine analytical workloads with numerical computing, for example in the context of large statistical analyses (expressed in Python or R). Furthermore, due to the growing popularity of data lakes [12], the interest in efficient solutions to analyze text-based data formats such as CSV and JSON is increasing even further.

At its core, the SQL language support in Spark relies on a managed language runtime – the Java Virtual Machine (JVM) – and on query compilation through so-called *whole-stage* code generation [7]. Whole-stage code generation in Spark SQL is inspired by the data-centric produce-consume model introduced in Hyper [23], which pioneered pipelined SQL compilation for DBMSs. Compiling SQL to optimize runtime performance has become common in commercial DBMSes (e.g., Oracle RDBMS [28], Cloudera Impala [42], PrestoDB [40], MapDB [38], etc.). Unlike traditional DBMSes, however, Spark SQL compilation does not target a specific data format (e.g., the columnar memory layout used by a specific database system), but targets *all* encoding formats supported by the platform. In this way, the same compiled code can be re-used to target multiple data formats such as CSV or JSON, without having to extend the SQL compiler back-end for new data formats. Thanks to this approach, 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 its source (e.g., a JSON file); in a second step, the compiled SQL code is executed over in-memory data. Such appealing modularity impairs performance, since the parsing step has to be executed in library code, which is not specialized for a specific query, in contrast to the generated code.

In this paper, we introduce a new approach to SQL query compilation for Spark that outperforms the state-of-the-art Spark SQL code generation with significant speedups of up to 4.4x on CSV and up to 2.6x on JSON data files. Our SQL code compilation is based on *dynamic* code generation, and relies on the intuition that the compiled query code should leverage static and runtime knowledge available to Spark as much as possible. Specifically, our SQL code generation (1) integrates data access (i.e., de-serialization) with data processing (i.e., query execution), minimizing de-serialization costs by avoiding unnecessary operations; (2) makes speculative assumptions on specific data properties (e.g., on the observed numeric types) to enable efficient "in situ" data processing, and (3) leverages runtime profiling information to detect when such speculative assumptions no longer hold, invalidating relevant compiled code, and generating new machine code accordingly. We have implemented our SQL compiler for Spark targeting two popular data for-

Proceedings of the VLDB Endowment, Vol. 13, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3377369.3377382

```
# Load the JSON data from a file
data = spark.read.json("orders.json")
# Create a temporary view over the data
data.createOrReplaceTempView("orders")
# Sum order prices
result = spark.sql("""
  SELECT SUM(price)
  FROM orders
  WHERE shipdate
    BETWEEN date '1994-01-01'
    AND date '1994-12-31'
""")
# print the query result to the console
result.show()
# the result is a valid Python object: can be
    used in other computation
doSomethingWith(result)
```

Figure 1: Example of a Spark Python application executing a simple SQL statement on JSON data.

mats, namely CSV and JSON. The reason for targeting textual data formats rather than more efficient binary formats such as Apache Parquet is their great popularity: the goal of our work is to show that by combining efficient data access and speculative runtime optimizations, the performance of textual data formats can be significantly increased.

This paper makes the following contributions:

* We describe a new SQL compiler for Spark which leads to speedups of up to 4.4x.
* Our code generation combines data access (i.e., data de-serialization) with data processing (i.e., the actual query execution). The generated code relies on the Truffle [43] framework, which is used to dynamically generate efficient machine code.
* Our code generation leverages speculative specializations to implement efficient predicate execution. We describe how common operations such as string comparisons or date range checks can be optimized using speculative assumptions.

This paper is structured as follows. In Section 2 we introduce code generation in Spark SQL and describe how code generation is used to optimize SQL query execution. In Section 3 we provide an overview of our novel compilation approach, detailing its main components. In Section 4 we present a detailed performance evaluation. Section 5 discusses related work, and Section 6 concludes this paper.

2. BACKGROUND: CODE GENERATION IN SPARK SQL

Spark SQL [2] allows data scientists to execute SQL queries on top of Spark. Unlike a traditional DBMS, SQL queries in Spark are executed by the Java Virtual Machine in a distributed cluster. To improve execution performance, Spark compiles SQL statements to executable Java classes, which are responsible for data loading as well as for the actual query execution. 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 run in a distributed way, using internal Spark abstractions such as Resilient Distributed Datasets [46] (RDDs).

An example of a Spark Python application performing a simple SQL query over a JSON file is depicted in Figure 1. The concise size of the code is a good indicator of Spark SQL's convenience for data analytics. Behind the scenes, Spark SQL (1) reads a potentially very large (and distributed) JSON file in parallel, (2) converts its content to an efficient in-memory representation, (3) generates Java code to perform the actual query execution, and (4) orchestrates a cluster of computers to send and collect all required data from and to the Python language runtime executing the application. Input-data parsing and code generation can affect performance as much as other runtime aspects such as data-parallel execution, I/O orchestration, workload distribution, etc.

```
class GeneratedBySpark {
  public void compute(Data input) {
    while (input.hasNext()) {
      // parse the next JSON input data
      Row row = input.parseNext();
      // materialize the needed data
      UTF8String date =
          row.getUTF8String("shipdate");
      // 1st date comparison
      if (date.compareTo('1994-01-01') < 0)
          continue;
      // 2nd date comparison
      if (date.compareTo('1994-12-31') > 0)
          continue;
      // accumulate value 'price' as result
      double price = row.getDouble("price");
      accumulate(price);
    }
  }
}
```

Figure 2: Java code produced by Spark for the SQL query of Figure 1.

The Java code that Spark generates at runtime for the example query is depicted in Figure 2[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 JSON 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 JSON files, on CSV 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. Specifically, the code in Figure 2 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 JSON parser) that consumes the *entire* body of the input data. This is potentially a significant waste of resources, since parsing each single JSON element in a very large file means allocating temporary, short-lived

[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.

objects (one object for each JSON value) in the JVM heap memory space. Short-lived values are not always needed to execute the entire query: depending on projectivity and selectivity, 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 (i.e., the date range in our example) has a *generic* implementation. I.e., they do not take into account the specific nature of the query, and simply rely on calls into the Spark core runtime library to implement operations such as date comparisons, etc. As shown in the previous example, this is a missed optimization opportunity.

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. In the following sections, we will describe how runtime specialization can be used to implement *speculative* optimizations to avoid the aforementioned limitations, leading to significant performance improvements for data formats such as JSON and CSV.

3. DYNAMIC SQL QUERY COMPILATION

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. In contrast, our approach to SQL compilation is *dynamic*: after an initial compilation of the query to machine code, the compiled query code has the ability to perform runtime profiling, and modify the behavior of the query execution (for example, by re-compiling the machine code responsible for certain SQL predicate evaluations to a more efficient version). Static compilation has the main limitation that runtime information cannot be exploited by the generated code. Conversely, dynamic compilation allows for more precise optimizations that can take into account any aspect of a query execution, triggering code optimization and de-optimization accordingly. This is achieved by means of the Truffle [43] framework. Truffle is a language implementation framework enabling the runtime generation of efficient machine code via runtime specialization and partial evaluation [13]. The key intuition behind the concept of specialization is that certain operations can be performed more efficiently when favorable runtime conditions are met. As an example, consider the implementation of the lookup operation in a hash map: general-purpose implementations (e.g., in `java.util.HashMap`) have to take into account aspects such as the type of keys and values, key collisions, internal storage size, etc. Conversely, a "specialized" version for, e.g., a map known to have only numeric keys and a small, bounded, number of values, could be implemented more efficiently by *specializing* lookup operations using a more efficient hashing function. Although simple and naive, this is an example of optimizations that are commonly performed by managed language runtimes, which produce specialized code for certain language operations (e.g., property reads in languages such as JavaScript or Python), and execute such code as long as the specialized operation is not *invalidated* by other runtime events (e.g., a non-numeric key value being added to our example map). Our new SQL code generation leverages runtime specialization in two key aspects of a query evaluation:

1. *Data access.* Different data formats (e.g., JSON and CSV) can be de-serialized in different ways. Specializing the SQL execution code to take into account certain aspects of the data serialization process can lead to a more efficient input-data de-serialization.
2. *Predicate execution.* Each predicate in a SQL statement (e.g., date range in the example of Figure 2) is *not* specialized for the specific predicate. Rather, general-purpose operations such as string or numeric comparisons are used regardless of the *properties* of the data being processed.

As we argue in this paper, applying specialization to these two operations can lead to significant performance benefits. In the following subsections we provide a detailed explanation about how such specialized execution can be performed at runtime in Spark SQL.

3.1 Dynamic Code Generation

Rather than generating static Java classes, our SQL code generation approach is based on the generation of *dynamic* runtime components that have the ability to influence how code is optimized at runtime, as well as the ability to profile the query execution, de-optimizing and re-optimizing machine code as needed during query execution. This is achieved by generating Truffle *nodes* as the result of SQL query compilation. Differently from standard Java classes, Truffle nodes have access to a set of *compiler directives* [26] that can be used to instruct the VM's just in time (JIT) compiler about runtime properties of the generated machine code [44]. When executed on the GraalVM [45] language runtime, the JIT compiler [9] can compile Truffle nodes to machine code capable of dynamic optimization and de-optimization. Contrary to static code-generation approaches such as Spark's code generation (or similar ones such as LLVM-based SQL query compilers [23, 5, 22]), generating Truffle nodes has the advantage that the generated code has the ability to monitor runtime execution, and optimize (or de-optimize) machine code when needed, taking into account runtime information such as, e.g., observed data types or branch probabilities. This is a key difference compared to the static SQL compilation based on "data-centric" code generation [23].

By leveraging Truffle APIs, our SQL query compiler can generate code that performs speculative optimizations based on certain runtime *assumptions*. Such assumptions may rely on specific runtime properties of the data being processed. For example, the generated code may *assume* that all years in fields of type `Date` have length of exactly four digits, and perform date comparisons between two years by comparing only some digits. As soon as a date with a year with more than four digits is found, the machine code performing the date comparison can be *invalidated* and replaced with *generic* machine code performing year comparisons on all possible values. This process is called *de-optimization* and can be performed by our generated code for arbitrary runtime assumptions. Speculative optimizations, their invalidation, and re-compilation to different machine code allow the generated code to adapt to the properties of the data. An overview of the overall SQL code generation implemented in our approach is depicted in Figure 3.

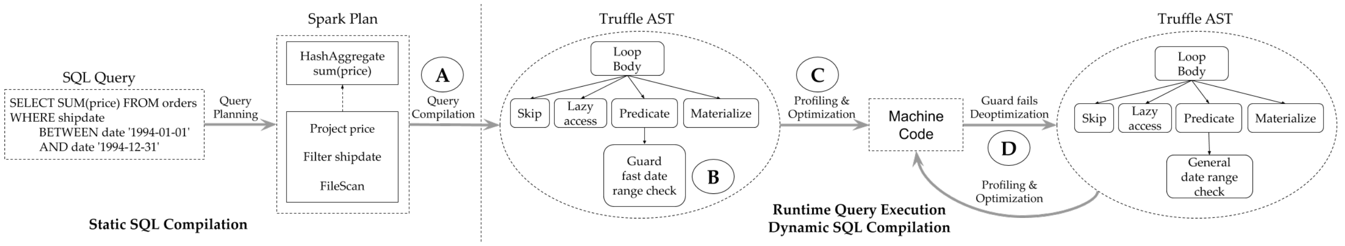


Figure 3: SQL Dynamic Compilation

With our approach, query compilation and execution take place in the following way:

1. When a new query is submitted to Spark SQL, its query plan is analyzed, and a set of Truffle nodes that can execute the given query is generated. An overview of this process is depicted in Figure 3 (A).
2. When the generated Truffle nodes are executed, they may rewrite themselves to take into account certain runtime assumptions. Specifically, they replace themselves with a more specialized version that is capable of executing only a given special-purpose version of an operation (e.g., predicate evaluation). This is depicted in Figure 3 (B).
3. During execution, Truffle nodes are compiled to machine code by the Graal compiler (Figure 3 (C)).
4. If a speculative assumption made during compilation is invalidated by unexpected runtime events, the compiled code is invalidated, *de-optimized*, and the corresponding Truffle nodes are replaced with generic implementations that do not rely on the failed runtime assumption. Query execution might re-profile the generic code to attempt the creation of new optimized code. This is depicted in Figure 3 (D), where a specialized Truffle node for a given predicate evaluation is replaced with a general-purpose one.

By generating Truffle nodes rather than plain Java classes, we generate code that can specialize for given runtime conditions and generate machine code that is specialized with respect to such conditions. Since in this work we are optimizing data access operations and predicate execution, our approach implements the process described above only for the leaf nodes in the query plan generated by Spark, i.e., a file-scan operator with projections and pushed-down predicates. Our optimizations change the internal implementation of the leaf nodes, but they do not alter their interface (i.e., an iterator of rows). In this way, even if other query plan operators are not affected by our code generation, and their generated code is produced using the default Spark runtime, our generated classes are perfectly integrated in the whole Spark compilation pipeline. Our code-generator relies on two different types of Truffle nodes that both contribute to the evaluation of a SQL query, namely (1) nodes to access and de-serialize data, and (2) nodes to evaluate predicates. In the following sections we describe how both node types contribute to query execution in our system.

3.2 Specializing Data Access (Spark-SDA)

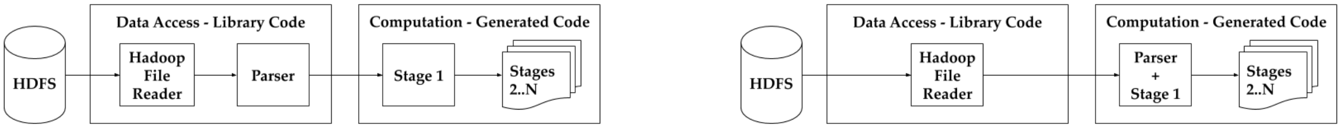
As Figure 2 shows, the code generated by Spark SQL performs a runtime call to a general-purpose data de-serializer for each entry in the input dataset. In the case of a line-delimited JSON file, this corresponds to a call to a JSON parser for each input line (i.e., `input.parseNext()` in Figure 2). Using a general-purpose de-serializer corresponds to a potential performance limitation due to the increased data scan and memory pressure costs. To overcome these inefficiencies our SQL compiler replaces the general-purpose parser used by Spark with multiple specialized parsers that are aware of the underlying format. As shown in Figure 4, our approach achieves a tight integration of data access operations with data processing, replacing the Spark general-purpose parser with Truffle nodes that can perform incremental and selective parsing of the input data. With such an approach, the code generated by our SQL compiler can avoid unnecessary conversion steps by directly de-serializing only the required subset of fields from raw input bytes to their equivalent Java types. Moreover, the generated code can parse input data incrementally and only *if* needed by the query, i.e., it is able to discard an entire row before fully parsing it, when it finds the first field which does not pass the query predicate. From now on, we call this optimization as Specialized Data Access (Spark-SDA).

Generating machine code that performs not only query evaluation but also input data de-serialization can be considered as an *enabling factor*, as it allows for further speculative optimizations during predicate evaluation, as we will further discuss in Section 3.3. In the following subsections we describe how our SQL code generator creates Truffle nodes that are capable of such incremental lazy parsing.

3.2.1 Specialized Speculative Data De-serialization

We have implemented specialized Truffle nodes capable of parsing CSV and JSON line-delimited files. Our 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 order in which fields are read during query evaluation should follow the order in which the data is actually consumed.

By specializing a CSV de-serializer for a specific query, the code-generation not only has access to the subset of the fields that will be processed by the query; it also has access to the *order* in which each value will be read during query evaluation. 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 5. To execute the example



HDFS
Data Access - Library Code
Hadoop File Reader
Parser
Computation - Generated Code
Stage 1
Stages 2..N
HDFS
Data Access - Library Code
Hadoop File Reader
Computation - Generated Code
Parser + Stage 1
Stages 2..N

Figure 4: Comparison between original Spark (left) and Spark-SDA (right).

query of Figure 1, 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 field `price` so that it can be retrieved later (if the predicate passes), (3) evaluating the predicate on field `shipdate`, and (4) materializing the value of field `price`.

The same optimization can be applied in the context of JSON de-serialization. However, values in JSON are not necessarily always declared in the same order, and it is possible for a given value to appear in different positions in two different JSON objects. As a consequence, data deserialization also needs to take into account such potential different ordering of key-value pairs.

In this scenario, access to JSON data can be optimized by the generated code using a speculative approach. Specifically, the SQL processing code can be created based on the assumption that *most* of 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. Such speculative de-serialization can be performed efficiently, generating optimized machine code assuming that all JSON objects have the same structure; if the assumption does not hold for one input JSON object (i.e., its fields do not match the expected order) the generated machine code is de-optimized.

```
|id:num|price:decimal|shipdate:date|.other fields.|
|1     |9.95         |1933-03-01   |..............|
```

Figure 5: Example input value and schema for the CSV values in the example.

3.2.2 Parsing Nodes Generation

With specialized data access, the machine code responsible for query execution will parse input data incrementally and only when needed for query execution, our SQL compiler generates a set of specialized Truffle nodes capable of various parsing operations. Such data de-serializers correspond to four main operations, each of which is implemented with a dedicated Truffle node:

* *Skip* nodes: Skips a data value without performing any data conversion.
* *Lazy data-access* nodes: Stores the initial position of a field and its length in integer variables to retrieve the value in the future.
* *Data-materialization* nodes: Materializes a field value by creating a string object from the original byte array, using positions computed during lazy-data-access operation.
* *Predicate*: Evaluates a given predicate.

Each generated Truffle node can be considered a basic operation contributing to SQL query evaluation. Algorithm 1 shows the procedures implemented in Spark-SDA that, given a query-plan leaf and the underlying data schema, generates Truffle nodes with both data-parsing and data-processing operations. This algorithm is implemented by our code-generator for both CSV and JSON data sources. The only difference is that for JSON we have to manage the case where the speculative assumption does not hold, i.e., an input row does not match the expected schema. More precisely, our code generator for JSON data sources implements the following variations:

* The source code generated by our Spark-SDA algorithm is wrapped in a new Truffle node.
* We invoke the original Spark code generator and the generated source code is wrapped in a second Truffle node.
* Truffle nodes for lazy data-access and skip operations are extended by generating a matching function which checks that the current field matches the expected one.
* If the matching function fails, the speculatively compiled node is de-optimized and replaced with the general node containing the code generated by Spark.

Algorithm 1 Code-generation Algorithm

```
procedure CodeGen($Projections, Predicates, Schema$)
    $F \leftarrow$ set of projected and filtered fields
    $n \leftarrow \max_{f \in F}(Schema.index(f))$
    for $i \leftarrow 0, n$ do
        $f \leftarrow Schema.fields[i]$
        if $f \notin F$ then
            emitCode(skip)
        else
            emitCode(lazy-data-access(f))
            $P \leftarrow$ PredicatesAtIndex($i$)
            for all $p \in P$ do
                PredicateCodeGen($p$)
            end for
        end if
    end for
    for all $f \in F$ do
        emitCode(data-materialization(f))
    end for
end procedure

procedure PredicateCodeGen($predicate$)
    for all $f \in fields(predicate)$ do
        emitCode(data-materialization(f))
    end for
    emitCode(spark-codegen(predicate))
end procedure
```

Since *data-materialization* nodes may be emitted multiple times for the same field (e.g., if a field is involved in more than one predicate), our code generator emits the node code only once for each field, subsequent calls simply return the empty string. Note that we left the procedure PredicatesAtIndex undefined, our approach is to evaluate each

```
// skip irrelevant field 'id'
skip()
// store position of column 'price'
pos_price = lazyAccess()
// read 'shipdate'
pos_shipdate = lazyAccess()
shipdate = materialize(pos_shipdate)
// check predicate
if (shipdate.compareTo('1994-01-01') < 0)
    continue;
if (shipdate.compareTo('1994-12-31') > 0)
    continue;
// all good: save the result
price = materialize(pos_price)
accumulate(price);
```

Figure 6: Pseudo-code produced by Spark-SDA for the query in Figure 1.

predicate as soon as possible, i.e., just after all fields involved in a predicate have been scanned by the parser. Our approach relies on the intuition that parsing is the most expensive operation, so evaluating a predicate as soon as all fields are available reduces the time spent on data parsing. The current implementation of PredicatesAtIndex returns the sequence of predicates which can be executed at a given field position, i.e., a predicate p is contained in PredicatesAtIndex(i) if and only if:

$$\max_{f \in fields(p)} (Schema.index(f)) = i$$

However, the predicate evaluation order used by our approach may not be the best one, i.e., it may be more efficient to postpone the evaluation of a predicate after another one, which involves a field with higher position. Changing the predicate evaluation order with our approach requires only to change the procedure PredicatesAtIndex so that it takes into account an alternative order. Such alternative predicate evaluation orders could be provided, e.g., manually by the user or automatically by a cost-based optimizer which takes into account the cost of reading raw data.

As an example, consider two predicates p_1 and p_2, where p_1 involves only the field at position i and p_2 involves the field at position j (with $i < j$). With our approach p_1 is evaluated once the parser reaches the i-th field and p_2 once it reaches the j-th field. Depending on the cost of evaluating such predicates, their selectivities, and the cost of performing a scanning operation from the i-th field to the j-th field, it may be more efficient to postpone the evaluation of p_1 after p_2. To postpone the evaluation of p_1 after p_2 the procedure PredicatesAtIndex should return an empty sequence given index i and $[p_2, p_1]$ given index j. With such different return values of PredicatesAtIndex, our code-generator would generate the expected code without further modifications, that is: the generated code would first store the position of the i-th field, then scan until the j-th field, evaluate p_2 and, if p_2 passes, evaluate p_1.

The pseudo-code generated using our Spark-SDA optimization for the query in Figure 1 can be found in Figure 6, where the main loop body of the same query is depicted.

3.3 Specializing Predicates (Spark-SP)

Our SQL code generation produces Truffle nodes that specialize themselves based on runtime assumptions. By generating Truffle nodes that can perform selective parsing on raw data, the Truffle nodes responsible for predicate evaluation can perform aggressive speculative optimizations that can be used to evaluate predicates operating on raw data in most cases. Consider the code that executes the date range check shown in Figure 6. This code corresponds to the evaluation of the SQL predicate `‘1994-01-01’ <= shipdate <= ‘1994-12-31’` from the query 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 a heap-allocated Java object. This operation introduces extra conversion overhead as well as one more memory copy. Moreover, 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. 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`.

The basic principle of specialization can be applied to the evaluation of predicates as well. Specifically, predicate evaluation can be performed more efficiently by leveraging the static information available at SQL compilation time, combined with runtime information on the actual observed data types. In this example, such information includes the data type (i.e., `date` in this case) and the expected operations to be performed (i.e., a date range check in our example).

The constant information can be leveraged by the SQL compiler to perform the predicate execution more efficiently by specializing the generated code for the actual predicate to be evaluated, and by performing the expected operation directly on the raw input data. That is, the compiler can generate machine code capable of performing the date comparison (1) without allocating new objects nor pushing primitives on the stack, and (2) evaluating each condition (i.e., `‘1994-01-01’ <= shipdate <= ‘1994-12-31’`) 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, and might fail for certain values observed at runtime (e.g., a date field may not match the expected date format). Therefore, the compiler shall generate code that checks for a given condition *speculatively*, i.e., under a given assumption, and falls back to a default implementation at runtime, if the assumption does not hold for a certain value.

In our Spark SQL compiler, we have implemented specialized predicate evaluation for a number of common SQL operations, ranging from numeric comparisons to date comparisons, to string equality and inclusion. In particular, we implemented a specialized version of all the predicates that Spark can push down to data sources [35]:

* Relational operators ($<, \leq, >, \geq, =, \neq$) for numerical, date and string datatypes.
* String operators: set inclusion and variant of SQL `LIKE` operator:
  - startsWith (`column LIKE "constant%"`)
  - endsWith (`column LIKE "%constant"`)
  - contains (`column LIKE "%constant%"`)

```
skip()
pos_price = lazyAccess()
pos_shipdate = lazyAccess()
// check predicate and get next cursor position
cursor = dateRangePredicate(pos_shipdate)
if(cursor == -1) continue;
price = materialize(pos_price)
accumulate(price);
```

Figure 7: Pseudo-code produced by Spark-SP for the query in Figure 1.

All specializations are implemented in presence of statically known literal values, we implemented relational comparisons between two variable values (i.e., columns) only for the date datatype. Note that a predicate which involves expressions among literal values is considered a literal value, too, e.g., a predicate like `t.a = '0.' + '07'` is statically evaluated by the Catalyst optimizer [8] and once pushed down appears as `t.a = '0.07'`.

To extend our SQL compiler with the Spark-SP optimization we used the code-generation Algorithm 1, with the following variations in the procedure PREDICATECODEGEN:

* We generate a Truffle node that can evaluate the predicate on the raw data with optimistic assumptions.
* We generate a second Truffle node that evaluates the predicate using the procedure PREDICATECODEGEN shown in Algorithm 1 (i.e., such node contains our data-materialization operations and the generic predicate evaluation code generated by Spark).
* If the optimistic assumptions hold, we execute the predicate with the first node, returning the index of the next field if the predicate passes or a sentinel value (i.e., `-1`) if the predicate does not pass.
* Otherwise, the speculative node is de-optimized and replaced with a general-purpose one.

The pseudo-code generated using our Spark-SP optimization for the query in Figure 1 can be found in Figure 7, where the main loop body of the same query is depicted.

In the following sections we detail the Truffle nodes used by our code generation for the most notable predicates.

3.3.1 *Numeric Comparisons*

The general idea behind leveraging specialization for SQL predicate evaluation is that a fast-path condition should be checked without incurring the runtime overheads associated with the slow-path (i.e., the original, generic, predicate evaluation): if most of the rows do not pass the predicate evaluation, an efficient fast-path can result in significant speedups. For numeric data types, such a fast-path approach could be implemented by specializing the predicate code taking into account the constant values in a query.

For integer fields (both int and long) we speculatively assume that the character sequence does not start with zero (i.e., the integer is not left-padded), so that we can implement a length-based comparison. For decimal fields we speculatively assume that the number of decimal digits respects the given format.

3.3.2 *String and Date Comparisons*

Like with numeric comparisons, string and date predicates can be implemented by means of runtime specialization based on the constant values in the SQL query. Similarly to numeric predicates, the code produced by our SQL code generation is composed of a specialized fast-path operation that takes into account the specific constant values of a given query (e.g., a constant date) and a slow-path to be used whenever the fast-path cannot be executed with confidence. In the context of date and string comparisons, the operators that our code generation can specialize are essentially string comparisons, string subset scans, and date comparisons. An example of specialized code for the latter is depicted in Figure 8 where a *date* literal is compared against constant values in our running example of a SQL query. As the figure shows, the fast-path is implemented by first checking that the input data has the same (expected) length; if successful, the entire date can be compared using a byte-by-byte comparison rather than allocating the actual date object on the JVM heap. As the figure shows, for date fields we speculatively assume that the date respects the given format (or the default one, i.e., `yyyy-MM-dd`).

```
// condition: 1994-01-01 <= date <= 1994-12-31
// input: int from (initial field position)
const lower = byte[] {'1', '9', '9', '4', '-',
    '0', '1', '-', '0', '1'};
const upper = byte[] {'1', '9', '9', '4', '-',
    '1', '2', '-', '3', '1'};
boolean sameLength = data[from+10]==DELIMITER;
boolean fastPath = sameLength
                && data[from+4] == '-'
                && data[from+6] == '-';
if (fastPath) {
  for(int i=0; i < upper.length; i++) {
    if (data[i+from] < lower[i]) return -1;
    // the predicate does not pass
    if (data[i+from] > upper[i]) return -1;
    // the predicate does not pass
  }
  return from + 11; // next cursor position
} else {
  // unexpected format: full comparison
  return slowPath(data);
}
```

Figure 8: Pseudo-code for predicate `1994-01-01 <= shipdate <= 1994-12-31`.

Like the example in the previous section, the code generated for this specific predicate evaluation can be considered a partially evaluated version of a general-purpose date comparison operation that is specialized for the input date range defined in a given SQL query. Like in the previous example, our code generation leverages Truffle nodes: whenever the fast-path condition is invalidated (e.g., a date with a year with more than four characters is found) the Truffle node is invalidated and replaced with the default, general-purpose date comparison operation of Spark.

3.4 Integration in Spark SQL

Our implementation does not require any modification to the Spark source code. Indeed, our code generation is an external plug-in that can be integrated into any unmodified Spark runtime. Our implementation relies on the experimental Spark SQL API [34], which allows developers to customize the query planning phase.

While working on the implementation of our SQL compiler, we noticed that Spark splits the `BETWEEN` operator into a conjunction of two predicates (namely, less-than and greater-than). We consider this approach another missing optimization opportunity, which we overcome within our Spark-SP optimization. To this end, our implementation analyzes all pushed-down predicates; whenever it finds a conjunction of predicates in the form `X < c AND c < Y` (where `c` is a column and both `X` and `Y` are constant values), our compiler merges both conditions into a single predicate. In this way, during the byte-by-byte comparison implemented within the predicate fast-paths, we are able to execute both predicates in a single pass, as shown in Figure 8.

Although we implemented our optimizations in Spark, since they affect the plan leaf nodes for data access and pushed-down predicate execution, the same approach can be implemented with other data processing systems based on SQL-compilation to Java bytecode which allow users to execute queries on raw data files (e.g., Apache Flink).

4. PERFORMANCE EVALUATION

In this section we evaluate the performance of our new SQL compiler for Spark using the TPC-H benchmark [41], comparing our approach against an unmodified Spark SQL runtime [7]. We evaluated our optimizations in two different settings, namely, on a single multicore machine and on a distributed cluster. The evaluation on a single machine has the goal of assessing the performance of our SQL code generation in isolation from I/O effects, evaluating the performance of our compilation techniques in an ideal scenario, i.e., where computation is CPU-bound. To this end, we run Spark in "local-mode", using a small dataset that fits entirely in the machine's main memory, and we selectively disable speculative predicate compilation (i.e., Spark-SP) to highlight the performance impact of the different optimizations presented in Section 3. The evaluation on a distributed cluster has the goal of demonstrating the performance of our SQL compiler in a realistic Spark configuration, to show that even in the presence of I/O pressure, our new SQL compiler achieves significant speedups. Moreover, to better characterize the performance benefits of our optimizations we evaluated them with micro-benchmarks and we analyzed the memory pressure and the Spark file-reader performance.

In the following subsections we first introduce the configurations of the single multicore machine and the cluster used for our experiments, then we compare the results of our approach against an unmodified Spark runtime, reporting speedup factors of the end-to-end execution time for each query in the TPC-H benchmark. Finally, we evaluate our optimizations with micro-benchmarks and we show the results of the analyses on the memory pressure and the Spark file reader.

4.1 Experimental Configuration

Software configuration. The operating system is a 64-bit Ubuntu 16.04 on both the single machine and the cluster nodes. The baseline for our experiment is an unmodified Spark SQL 2.4.0 runtime with input data generated by the original TPC-H dbgen [41] tool. Moreover, each benchmark is executed using the GraalVM EE 1.0.0-rc13 Java Virtual Machine. Each experiment is executed 10 times; in the figures we present the arithmetic mean of the 10 measurements as well as 95% confidence intervals (error bars), or the related speedup factors, evaluated on the arithmetic mean.

The JVM for our benchmarks is GraalVM. This is motivated by the fact that our SQL compiler relies on the Truffle framework and on the Graal compiler (both included in GraalVM) for efficient compilation to machine code. The Graal compiler offers performance comparable to C2, i.e., the default optimizing compiler in the HotSpot JVM. To ensure that GraalVM (i.e., the baseline used in our experiments) has performance comparable to that of popular JVMs used in real-world Spark deployments, we executed the TPC-H benchmark running an unmodified Spark SQL runtime on both GraalVM and HotSpot (Oracle Java JDK, version jdk1.8.0_192). The results depicted in Figure 9 show that both VMs have equivalent performance, therefore ensuring that all the speedup factors that we report are obtained against a realistic, high-performance baseline.

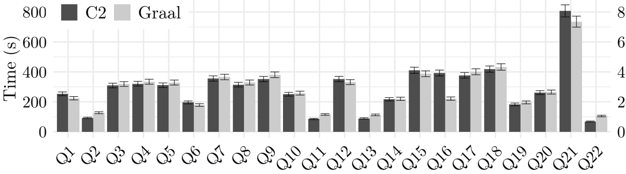


Figure 9: TPC-H performance. Comparison between HotSpot's C2 and GraalVM's Graal JIT.

Single machine configuration. The multicore machine we use in the first setting has 128 GB of RAM and two Intel Xeon E5-2680, each of them with 8 cores (@2.7GHz). To be sure the data fits in main memory, we use a dataset of scale factor 30 (30 GB dataset of raw data) for CSV benchmarking and scale factor 10 (27 GB dataset of raw data) for JSON benchmarking.

Cluster configuration. We have executed all cluster experiments on a block of 8 nodes in the Data Center Observatory (DCO) [11]. Each node has 128 GB of RAM and two AMD Opteron(TM) Processor 6212, each of them with 8 cores (@2.6GHz). All datasets are stored in Hadoop File System (HDFS) [32] version 3.1.2, among 7 slave nodes, since we reserve one node as the Spark master node. For the cluster configuration, we use a dataset of scale factor 300 (300 GB dataset of raw data) for CSV benchmarking and scale factor 100 (270 GB dataset of raw data) for JSON benchmarking. The benchmark is executed using a Spark YARN [33] master node and 7 slave nodes, which are the same machines that run HDFS, such that Spark can take advantage of the node-local [36] data locality.

4.2 TPC-H - Single Machine

We evaluate the performance of our new code-generation approach on the well-established industry-standard TPC-H benchmark suite, using the CSV and JSON data formats. The goal of our experimental evaluation is to show how our code-generation technique affects the performance of end-to-end query execution times.

As discussed, our code generation benefits from two different techniques, namely Spark-SDA and Spark-SP. Depending on the query workload, the impact of the two techniques can be different. With the goal of precisely assessing the impact of each individual technique excluding other potential aspects (especially I/O) we first evaluate our SQL compiler on a single machine. To this end, we first measure the performance of end-to-end query execution time with Spark-SDA alone, and then we enable Spark-SP. Since the Spark-SP optimization is implemented on top of Spark-SDA, it is not

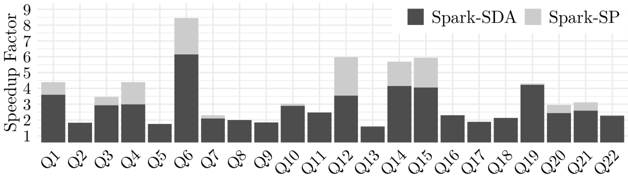


Figure 10: Query execution performance on a single multicore machine. CSV dataset (SF-30).

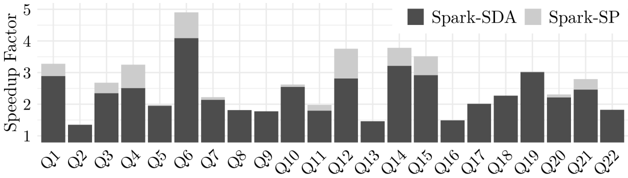


Figure 11: Query execution performance on a single multicore machine. JSON dataset (SF-10).

possible to evaluate it in isolation. Then, we evaluate the combined optimizations on a cluster, to show the speedup obtained by our approach in a more realistic configuration, running Spark on a cluster with a bigger dataset.

Our code-generation approach affects the so-called "physical operations" of a query, i.e., the leaves of the query plan generated by Spark (i.e., file scans with filter, and projection operations). As a consequence, we expect the impact of our novel code generation to be more significant on queries whose execution plan does not involve many joins or complex aggregations. Furthermore, executing Spark on a cluster, joins and aggregations involve "shuffle" operations, i.e., operations that require the communication of partial results among executor instances. Intuitively, queries that involve many shuffle operations are much more subject to I/O pressure, and the data transfer cost is often non-negligible [24].

Since it is well-known that shuffle operations are often one of the major bottlenecks in Spark SQL, the TPC-H queries have been categorized [6] by the amount of data transferred during the shuffle operations:

* shuffle-heavy: queries composed of many joins or aggregations require much network I/O to share partial results among the cluster nodes during computation. Queries 5, 7, 8, 9, 18, 21 are representative for this category.
* shuffle-light: conversely, queries with few joins or aggregations are more computation-intensive. Queries 1, 4, 6, 12, 13, 14, 15, 19, 22 are representative for this category.

Not surprisingly, our code-generation approach achieves better performance on queries that belong to the shuffle-light category, since network I/O is not a bottleneck.

For each benchmark, we present the results collected using both the CSV and the JSON data formats. Obtained individual speedups are depicted in Figure 10 for the CSV dataset, and in Figure 11 for the JSON dataset.

4.2.1 Spark-SDA Evaluation

The most important factor that impacts the speedup of Spark-SDA is predicate selectivity (since a low selectivity allows us to discard many rows, avoiding unnecessary de-serialization steps) and the position of filtered fields (i.e., if filters are applied to the initial columns of the CSV file, our approach can skip much more computation steps).

The figures show the following notable properties of our code generation:

* Shuffle-light queries with very low selectivity (i.e., 6, 12, 14, 15, 19) achieve speedups ranging from 3.55x (Q12) to 6.15x (Q6).
* Most of remaining shuffle-light queries (i.e., 1, 4, 22) achieve speedups ranging from 2.25x (Q22) to 3.6x (Q1).
* Overall, the lower speedup of Spark-SDA is 1.58x on CSV and 1.34x for JSON, and the average speedup is 2.8x for CSV and 2.31x for JSON.

As the benchmark shows, specialized data access leads to significant speedups with shuffle-light queries; this is expected, and suggests that queries where de-serialization is the major bottleneck benefit most from our Spark-SDA optimization.

4.2.2 Spark-SP Evaluation

The most important factor that impacts the speedup of Spark-SP is the data types involved in the predicates. In particular, predicates accessing `date` fields are often the most expensive ones. Another important factor is the time spent during the input tokenization, which is the reason for the different contribution of Spark-SP w.r.t. Spark-SDA alone for CSV and JSON, since evaluating the predicates on raw CSV or JSON data have intuitively the same cost (i.e., both of them are byte arrays containing UTF8-encoded strings). Indeed, accessing a JSON dataset involves roughly twice the number of tokenization operations (i.e., by accessing JSON data, not only the values have to be tokenized, but also the object keys).

The figures show the following notable properties of our code generation:

* Queries that involve predicates on date fields (i.e., 1, 6, 12, 14, 15, 20) benefit most from our approach, achieving a speedup up to 8.45x (Q6) compared to the baseline and up to 1.69x compared to Spark-SDA alone.
* Average speedup is 3.34x for CSV and 2.53x for JSON.

Since many queries have no pushed-down predicate on the largest table (e.g., queries 2, 5, 8, 9, 11, 16, 17, 18, 21, 22) they do not benefit from the Spark-SP optimization, which is the reason of the small average speedup improvement (3.35x) compared to the one obtained with Spark-SDA alone (2.8x). Furthermore, another important reason for such a small improvement is that when our optimizations are enabled the Spark file reader quickly becomes the query-execution bottleneck. Indeed, our experiments show that excluding the common time spent within the file reader, the speedup executing Q6 is 11.34x for Spark-SDA and 25.3x for Spark-SP, this analysis is explained in details in the next section.

4.3 TPC-H - Cluster

With the goal of showing the impact of our optimizations on a realistic Spark deployment, we have also evaluated our SQL code generator on a distributed cluster. The performance numbers for the TPC-H benchmark are depicted in Figures 12 and 13. As the figures show, the obtained speedups by executing Spark on a cluster are lower compared to the ones obtained on a single machine. This is expected, since our approach does not optimize the distributed

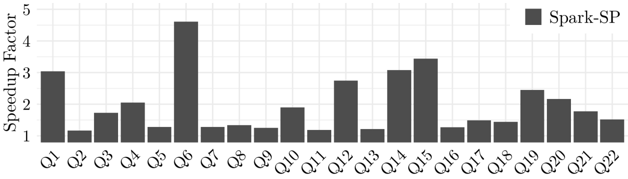


Figure 12: Query execution performance on an 8-nodes Spark cluster. CSV dataset (SF-300).

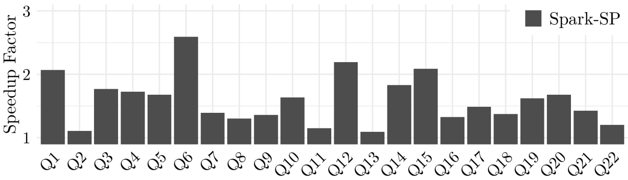


Figure 13: Query execution performance on an 8-nodes Spark cluster. JSON dataset (SF-100).

operations, but the leaves of query execution plans. Even if the speedups on a cluster are lower, they are still significant. Indeed, our SQL compiler obtains a speedup of up to 4.6x for CSV and 2.6x for JSON, whilst the average speedups are 1.93x for CSV and 1.6x for JSON.

4.4 Micro-Benchmarks

In this section we discuss a set of microbenchmarks we have designed to better highlight the impact of our code generation to query execution performance. All the micro-benchmarks discussed in this section are evaluated executing Spark with a single thread, to allow for fine-grained measurements of the performance of the generated code.

4.4.1 Experiment 1: Predicates Field Position

As discussed in Section 4.2.1, our code generation leads to better performance when predicates are evaluated on fields with lower index position. Conversely, when a field is located at the end of an input data row, the performance gain deriving from selective parsing is reduced, as the parser still has to scan the entire data. To highlight the performance impact of field positions on a query evaluation, we created a microbenchmark based on the TPC-H table `lineitem`, using the CSV dataset of scale factor 1. In this microbenchmark we created 16 copies of the table, each having the `l_shipdate` field at a different position. Then, we have evaluated the following query on all generated tables:

```
SELECT COUNT(*) FROM lineitem
WHERE l_shipdate < date '1990-01-01'
```

Since the query produces an empty result, the experiment measures reading and predicate time. Figure 14 shows the speedups obtained by our code generation for each generated table. As the figure shows, our code generation leads to higher speedups when the field is found at the first positions (i.e., 4.92x for Spark-SDA and 8.97x for Spark-SP). This is expected, as the greater the field position, the tokenization and fields skipping overhead increases. Moreover, we believe that by integrating a parser like Mison [20] with our approach we would not suffer from such penality, since by using Mison we would be able to "jump" to the field value without performing a scanner-based tokenization and fields skipping. However, as discussed in Section 5, integrating Mison with our approach would require to address practical challenges that we consider beyond the scope of this work, and that are related to the invocation of SIMD instructions from Java.

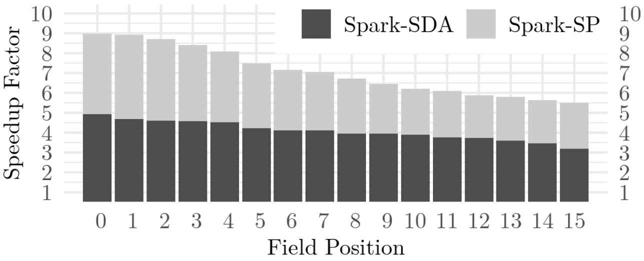


Figure 14: Execution performance, sensibility to field position. Single thread, CSV dataset (SF-1).

4.4.2 Experiment 2: Double vs Decimal Datatypes

As discussed in Section 4.2.2, the performance benefits of Spark-SP may vary depending on the datatypes involved in a predicate evaluation. In particular, Spark-SP provides better performance if the datatype is not converted to a Java primitive type by Spark, as the generated code may avoid object allocations and expensive comparison operations altogether. To highlight the impact of datatypes on predicate evaluation, we have created a version of TPC-H query 6 that uses decimal types and we have compared the Spark performance for the query against the default TPC-H query 6, which uses double values. For this micro-benchmark we used the CSV dataset with scale factor 10. Table 1 shows the query execution times of query 6 and the speedups obtained with our approach with both datatypes. As the table shows, execution times of both original Spark and Spark-SDA are much higher when using the decimal datatype, increasing from 178s to 236s with original Spark and from 28s to 46s with Spark-SDA, and the speedup factor of Spark-SDA over original Spark decreases from 6.36x to 5.13x. This is expected, since Spark-SDA uses the same conversion and comparison functions as original Spark, and the evaluation of such functions can easily became a bottleneck. On the other hand, Spark-SP does not suffer from such slowdowns, since most of the rows do not pass the predicates, only the few decimal values comprised in the rows which pass the predicate are actually converted. Indeed, using decimal datatype the query execution time with Spark-SP only increases from 20s to 21s, and its speedup compared to original Spark increases from 8.9x to 11.23x. This result highlights that specializing the generated code for a given data type leads to increased performance gains on top of more efficient data de-serialization.

Table 1: Q6 execution performance, comparing double and decimal datatype. Single Thread (SF-10)

Implementation	Time (double)	Speedup (double)	Time (decimal)	Speedup (decimal)
CSV	178s	-	236s	-
CSV-SDA	28s	6.36x	46s	5.13x
CSV-SP	20s	8.9x	21s	11.23x

4.4.3 Analysis 1: Memory Pressure

A reduced number of intermediate allocations should have a positive impact on the JVM's garbage collector. We evaluated the memory behavior of our code generation by collecting memory usage statistics using the Java Flight Recorder sampling profiler [25] during the execution of TPC-H query 6 with the CSV dataset of scale factor 10, running Spark with a single thread. Table 2 shows the number of Thread Local Area Buffer (TLAB) allocations[2], together with the total size of allocated objects, the number of garbage collector invocations and the total time spent during garbage collection. As the table shows, our approach greatly reduces the memory pressure. Indeed, original Spark allocated 111 GB in the Java memory heap, Spark-SDA reduces the allocated size to 13 GB, i.e., 8.5x less allocation size, and Spark-SP reduces the allocated size to only 600 MB, i.e., 22x less allocated size than Spark-SDA and 189x less allocated size than original Spark. We consider such reduced memory pressure an important result in a system like Spark, since it is known that in distributed data processing systems careful memory management is crucial [1].

Table 2: Query 6 - Memory Pressure. Single thread (SF-10), Java Memory Heap 60GB

Implementation	# TLABs	Size	# GC	GC Time
CSV	12 943	111 GB	255	1 235ms
CSV-SDA	603	13 GB	12	75ms
CSV-SP	122	600 MB	3	34ms

4.4.4 Analysis 2: File Reader Performance

As discussed in Section 4.1, for our microbenchmarks we have used a dataset size which fits in main memory, so that the operating system can cache the dataset content. Even with such configuration, we noticed that the file reader used by Spark (i.e., `HadoopFileLineReader`) became the major bottleneck when our optimizations were enabled. Consequently, we analyzed the file reader performances, comparing the total query execution time with the time spent within the file reader, and we evaluated the speedup factor of our approach after removing reading time, which does not vary depending on the implementation, since we have not optimized the file reader. With this analysis we executed Spark with a single thread, executing TPC-H query 6 on a dataset of scale factor 10 for both CSV and JSON sources.

As shown in Table 3, from our analysis it emerges that the query execution time is 178s for CSV and 237s for JSON, and the time spent within the `HadoopFileLineReader` class is 13.5s for CSV (7.6% of total execution time) and 33s (13.9% of total execution time) for JSON. With Spark-SDA enabled the total execution times are 28s for CSV and 54s for JSON, the percentage of time spent within the file reader is 48.3% for CSV and 61.1% for JSON, and the speedup factors of the execution time without file reading time are 11.34x for CSV and 9.71x for JSON. With both Spark-SDA and Spark-SP enabled, the total execution times are 20s for CSV and 44s for JSON, the percentage of time spent within the file reader is 67.5% for CSV and 75% for JSON, and the speedup factors of the execution time without file reading time are 25.3x for CSV and 18.54x for JSON.

Such analysis clearly shows that our optimizations push the bottleneck from data parsing to file reading, even in case the file has been cached by the operative system.

Table 3: Q6 execution performance, analysis of time spent in file reader class. Single Thread (SF-10)

Implementation	Time	Speedup	Reader	Time w/o Reader	Speedup w/o Reader
CSV	178s	-	7.6%	164.5s	-
CSV-SDA	28s	6.36x	48.3%	14.5s	11.34x
CSV-SP	20s	8.9x	67.5%	6.5s	25.3x
JSON	237s	-	13.9%	204s	-
JSON-SDA	54s	4.39x	61.1%	21s	9.71x
JSON-SP	44s	5.39x	75%	11s	18.54x

5. RELATED WORK

SQL query compilation in Spark plays a crucial role from a performance perspective, and dates back to the introduction of "whole-stage code generation" in Spark 2.0. To the best of our knowledge, no existing SQL code generation technique for Spark or comparable Big data systems relies on specialization, speculative optimizations, dynamic code generation and runtime de-optimization.

Flare [10] is another example of alternative SQL compilation in Spark. Based on the code-generation framework Lightweight Modular Staging [30], Flare can achieve impressive speedups, but -unlike our approach- it was specifically designed to operate on a single, shared-memory, multicore machine, and cannot execute Spark applications in a distributed cluster. Unlike our approach, Flare is a complete replacement of the Spark execution model, which avoids entirely network communication and the fault-tolerance. Moreover, certain Spark applications (e.g., those relying on user-defined functions) cannot be executed on Flare. Conversely, as discussed in Section 3.4 our code generation integrates directly with an unmodified Spark runtime, and can be used in all Spark execution modes (i.e., local mode, standalone, Yarn and Mesos clusters). Moreover, our new SQL compiler is completely transparent to the user, and does not impose any restriction on the applications that can be executed. Outside of the realm of Spark, SQL compilation is becoming pervasive in modern data processing systems, including several DBMSs [18]. One relevant example is ViDa [16], a query engine that allows one to execute queries on data lakes of raw data (e.g. CSV files). Similarly to our approach, ViDa integrates a specialized CSV de-serializer within the generated code, to reduce the parsing overhead by scanning tokens selectively and executing predicates as soon as the relevant column has been read. Besides ViDa, many other compilation techniques have been proposed, including JIT compilation from a bytecode format to dynamically-generated LLVM code snippets (e.g., in SQLite [17]) and an adaptive query execution system that dynamically switches from interpretation to compilation [19].

Differently from Flare, ViDa, and other "data-centric" SQL compilers, our code-generation approach relies on speculative dynamic optimization (and runtime de-optimization when speculative assumptions no longer hold). This is a fundamental difference in our compilation approach, and allows our compiler to aggressively optimize predicate execution, and to "replace" optimizations with less-specialized machine code when speculative assumptions no longer hold. On the

[2] TLAB is a memory area where new objects are allocated. Once a TLAB is full, the thread gets a new one. Therefore, the smaller the number of allocations, the better for the VM's Garbage collector.

contrary, the code generated for a SQL query in Flare, ViDa, and other systems is static C/LLVM code which is not dynamically de-optimized and re-compiled to account for the failures of speculative optimizations.

Adaptive Query Execution [37] is a framework developed by the Intel and Baidu big-data teams. Adaptive execution decreases the effort involved in tuning SQL query parameters and improves the execution performance by choosing a better execution plan and parallelism at runtime. Similar frameworks have been proposed in the literature (e.g., DynO [15], Rios [21]) to address the same problem, focusing on adaptively changing the join implementation during the query execution. Adaptive execution, DynO, and Rios implement adaptivity by updating the query plan at runtime, such approach requires to re-execute the full SQL static compilation pipeline (i.e., Java source generation and compilation to Java bytecode), incurring in non-negligible overhead. On the other hand, our approach implements adaptivity by meaning of dynamic recompilation and it completely avoid to re-generate sources and bytecode, since it de-optimizes and recompiles the existing bytecode.

Since a well-known major bottleneck of modern Big data systems is data de-serialization, several systems have focused on the problem of fast JSON de-serialization in Spark. Mison [20] is an example of such systems. It is based on the idea that predicates and projections can be "pushed down" to JSON parsing, an intuition that is shared with our code generation. Specifically, Mison relies on a positional index for the dataset values and on speculating over the value positions, and uses SIMD instruction to vectorize scanning operations. In contrast to our approach, Mison does not generate specialized code for the parser based on static knowledge (i.e., the data schema). The limitation of our approach compared with Mison is that our parser is subject to the position of queried fields, as discussed in Section 4.4.1. On the other hand, Mison approach is to speculate over the expected position of a field in the input byte array, rather than on the JSON object schema, like our approach. We believe that integrating Mison with our approach can help reduce the sensibility of our parser to the field positions.

Theoretically, Spark-SP optimization could be synergistic with Mison, i.e., we could invoke the Mison parser for each queried field and execute our speculative predicates on its result. Furthermore, after the training phase performed in Mison to generate the positional index ends, we could invoke a de-optimization and recompilation to inline the evaluated index, improving the generated machine code. The main challenge with such approach is that Mison requires SIMD instructions, and it is known that directly invoking them from Java is a challenging problem [31]. Mison implementation relies on the Java Native Interface (JNI) [27] to invoke the statically compiled Mison library from the JVM. Since it is known that JNI introduces an high overhead [14], Mison performs such invocation for each input file, instead of for each tuple, in that way the JNI overhead is alleviated. Unfortunately, such approach is not compatible with our optimizations, since the whole data access operation is performed within the Mison library.

Another relevant approach based on predicate push-down is Sparser [29]. The idea behind Sparser is to execute a pre-filtering phase over raw data which can drop rows that surely do not match certain SQL predicates. Generally, such a pre-filtering phase is not completely accurate, since if a row passes the pre-filtering it may still be dropped later by the actual filters. Hence, using Sparser, such a pre-filtering phase may avoid to parse a subset of data rows which is known not to pass a specific predicate. This idea is similar to our fast paths described in Section 3, but instead of integrating the pre-filtering phase within the Spark code generation, Sparser delegates it to a non-specialized (external) system. Furthermore, Sparser can apply its pre-filtering phase only to a small subset of SQL predicates, namely the exact string match, string inclusion, null checks, and their composition with logical AND and OR operators. Conversely, as discussed in Section 3.3, our Spark-SP optimization supports more complex predicates, like numerical and date comparison and range inclusion. Since Sparser introduces a pre-filtering phase and then uses an unmodified Spark runtime to execute the query on the filtered data, it can be easily integrated with our approach to obtain combined benefits.

Outside of the realm of SQL processing, FAD.js [3] is a runtime for Node.js that can parse JSON objects incrementally, leading to significant performance improvements. With our techniques, FAD.js shares the idea that data should be parsed only when needed. However, our approach is integrated within the SQL compilation of Spark SQL, whereas FAD.js is a runtime library for Node.js.

6. CONCLUSION

In this paper we introduced a new dynamic SQL compiler for Apache Spark. Our novel approach to SQL compilation relies on the intuition that data de-serialization and predicate evaluation should be specialized according to the input data format and other static or dynamic aspects of a SQL query. To this end, we have introduced two optimization techniques, namely Spark-SDA and Spark-SP, and we have evaluated them on Spark SQL workloads using CSV and JSON input data. Spark-SDA relies on the idea that input data de-serialization should be performed incrementally, leveraging parsing "steps" that are specialized for the given input data format. Spark-SP relies on the idea that a speculative "fast-path" predicate check can be performed very efficiently, without performing conversions to Java data types. Our novel SQL compiler outperforms the state-of-the-art Spark SQL compiler in most of the TPC-H queries, with speedups up to 4.4x for CSV and 2.6x for JSON on a distributed Spark cluster.

In future work, we plan to expand our novel code generation for Spark SQL to target Parquet and other columnar data formats. To this end, we will investigate how to extend our SQL compiler to invoke data-parallel SIMD instructions such as AVX2 and AVX-512 directly from Java, without incurring in the JNI overhead. By doing so, we would also be able to integrate existing SIMD-based components with our optimizations.

7. 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 VLDB reviewers for their detailed feedback and 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.

8. REFERENCES

[1] M. Armbrust, T. Das, A. Davidson, A. Ghodsi, A. Or, J. Rosen, I. Stoica, P. Wendell, R. Xin, and M. Zaharia. Scaling Spark in the Real World: Performance and Usability. *PVLDB*, 8(12):1840–1843, 2015.

[2] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[3] D. Bonetta and M. Brantner. FAD.js: Fast JSON Data Access Using JIT-based Speculative Optimizations. *PVLDB*, 10(12):1778–1789, 2017.

[4] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, pages 28–38, 2015.

[5] J. Chen, S. Jindel, R. Walzer, R. Sen, N. Jimsheleishvilli, and M. Andrews. The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database. *PVLDB*, 9(13):1401–1412, 2016.

[6] T. Chiba and T. Onodera. Workload characterization and optimization of TPC-H queries on Apache Spark. In *ISPASS*, pages 112–121, 2016.

[7] Databricks. Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop, 2019. `https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html`.

[8] Databricks. Spark SQL Adaptive Execution Unleashes The Power of Cluster in Large Scale, 2019. `https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html`.

[9] G. Duboscq, T. Würthinger, L. Stadler, C. Wimmer, D. Simon, and H. Mössenböck. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *VMIL*, pages 1–10, 2013.

[10] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf. Flare: Optimizing Apache Spark with Native Compilation for Scale-up Architectures and Medium-size Data. In *OSDI*, pages 799–815, 2018.

[11] ETH DCO. Welcome - Data Center Observatory — ETH Zurich, 2019. `https://wiki.dco.ethz.ch/`.

[12] H. Fang. Managing data lakes in big data era: What's a data lake and why has it became popular in data management ecosystem. In *CYBER*, pages 820–824, 2015.

[13] Y. Futamura. Partial Computation of Programs. *RIMS Symposia on Software Science and Engineering*, 1983.

[14] N. A. Halli, H.-P. Charles, and J.-F. Méhaut. Performance comparison between Java and JNI for optimal implementation of computational micro-kernels. *CoRR*, abs/1412.6765, 2014.

[15] K. Karanasos, A. Balmin, M. Kutsch, F. Ozcan, V. Ercegovac, C. Xia, and J. Jackson. Dynamically Optimizing Queries over Large Scale Data Platforms. In *SIGMOD*, pages 943–954, 2014.

[16] M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. *CIDR*, 2015.

[17] A. Kashuba and H. Mühleisen. Automatic Generation of a Hybrid Query Execution Engine. *CoRR*, 2018.

[18] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything You Always Wanted to Know About Compiled and Vectorized Queries but Were Afraid to Ask. *PVLDB*, 11(13):2209–2222, 2018.

[19] A. Kohn, V. Leis, and T. Neumann. Adaptive Execution of Compiled Queries. In *ICDE*, pages 197–208, 2018.

[20] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann. Mison: A Fast JSON Parser for Data Analytics. *PVLDB*, 10(10):1118–1129, 2017.

[21] Y. Li, M. Li, L. Ding, and M. Interlandi. RIOS: Runtime Integrated Optimizer for Spark. In *SoCC*, pages 275–287, 2018.

[22] P. Menon, T. C. Mowry, and A. Pavlo. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *PVLDB*, 11(1):1–13, 2017.

[23] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[24] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. Lu. Skyway: Connecting Managed Heaps in Distributed Big Data Systems. In *ASPLOS*, pages 56–69, 2018.

[25] Oracle. About Java Flight Recorder, 2019. `https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm#JFRUH170`.

[26] Oracle. com.oracle.truffle.api (GraalVM Truffle Java API Reference), 2019. `https://www.graalvm.org/truffle/javadoc/com/oracle/truffle/api/package-summary.html`.

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

[28] Oracle RDBMS. Database — Cloud Database — Oracle, 2019. `https://www.oracle.com/it/database/`.

[29] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia. Filter Before You Parse: Faster Analytics on Raw Data with Sparser. *PVLDB*, 11(11):1576–1589, 2018.

[30] T. Rompf and M. Odersky. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *GPCE*, pages 127–136, 2010.

[31] A. Stojanov, I. Toskov, T. Rompf, and M. Püschel. SIMD Intrinsics on Managed Language Runtimes. In *CGO 2018*, pages 2–15, 2018.

[32] Team Apache Hadoop. Apache Hadoop, 2019. `https://hadoop.apache.org/`.

[33] Team Apache Hadoop. Apache Hadoop 2.9.2; The YARN Timeline Service v.2, 2019. `http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/TimelineServiceV2.html`.

[34] Team Apache Spark. ExperimentalMethods (Spark 2.4.0 JavaDoc), 2019. `https://spark.apache.org/`

docs/2.4.0/api/java/org/apache/spark/sql/ExperimentalMethods.html#extraOptimizations().

[35] Team Apache Spark. spark/filters.scala at v2.4.0 apache/spark, 2019. https://github.com/apache/spark/blob/v2.4.0/sql/core/src/main/scala/org/apache/spark/sql/sources/filters.scala.

[36] Team Apache Spark. Tuning - Spark 2.4.0 Documentation, 2019. https://spark.apache.org/docs/latest/tuning.html#data-locality.

[37] Team Databricks. Spark SQL Adaptive Execution Unleashes The Power of Cluster in Large Scale, 2019. https://databricks.com/session/spark-sql-adaptive-execution-unleashes-the-power-of-cluster-in-large-scale.

[38] Team MapDB. MapDB, 2019. http://www.mapdb.org/.

[39] Team Parquet. Apache Parquet, 2019. https://parquet.apache.org/.

[40] Team PrestoDB. Presto — Distributed SQL Query Engine for Big Data, 2019. http://prestodb.github.io/.

[41] TPC. TPC-H - Homepage, 2019. http://www.tpc.org/tpch/.

[42] S. Wanderman-Milne and N. Li. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.*, pages 31–37, 2014.

[43] C. Wimmer and T. Würthinger. Truffle: A Self-optimizing Runtime System. In *SPLASH*, pages 13–14, 2012.

[44] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, and M. Grimmer. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. *SIGPLAN Not.*, pages 662–676, 2017.

[45] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Onward!*, pages 187–204, 2013.

[46] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, pages 2–2, 2012.

[47] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, pages 10–10, 2010.