Profiling Compiled SQL Query Pipelines in Apache Spark

Christian Stuart

January 21, 2020

Academic supervisor:	Peter Boncz, <peter.boncz@cwi.nl>
Daily supervisor:	Bogdan Ghit, <bogdan.ghit@databricks.com>
Host organisation/Research group:	Databricks, https://databricks.com/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Contents

Abstract

Users of Apache Spark™ regularly encounter the problem of suboptimal execution of SQL queries. A query optimizer, such as Spark's Catalyst optimizer, can often resolve potential problems in queries defined by the user, but these optimizations are bound by logical correctness. A such, a subomtimal query can often result in performance problems. For this reason, Spark provides the user with a user interface called Spark UI, that allows them to analyse problems with the runtime performance of their queries. However, in Spark version 2.0, a new data-centric execution model, based on ideas introduced in the HyPer SQL engine, was implemented. In this execution model, Spark employs Java code generation to compile operator pipelines in a SQL query plan into a single *for*-loop, resulting in highly efficient pipelines. A downside of this approach is that it obfuscates operator boundaries, and as a consequence, runtime observability within these pipelines is drastically reduced. Consequently, the Spark UI cannot give detailed information on the behavior of the individual operators of such a pipeline.

In order to create a tool that allows the user to analyse the runtime characteristics of the operators in their SQL queries, we explore different approaches for profiling compiled query pipelines in the JVM, with the aim of finding a profiling approach that can provide fine-grained and accurate information on individual operators in a compiled SQL query pipelines. Furthermore, a proposal is made for the definition and evaluation of the accuracy of such a profiler.

In this thesis, we introduce an approach of collecting fine-grained profiles of generated-code pipelines of SQL queries in Apache Spark, which allows users to break down the execution cost of these pipelines by individual operators. We achieve this by profiling at the level of native instructions, and using the JVM Tool Interface to aggregate these results at the level of lines of generated Java code. Furthermore, we show that this approach can be used to profile SQL query pipelines at a higher accuracy than existing JVM profiling approaches, without requiring functional changes to the generated source code, and finally we show that this approach can collect highly accurate profiles at an overhead of less than 10% for most queries, and can potentially obtain profiles at even lower overheads.

Chapter 1

Introduction

In this chapter, we will discuss the motivation of the research, and discuss the problem statement and research questions for this thesis.

1.1 Motivation

Apache Spark™ [33] is a data analytics framework, developed to efficiently perform large-scale distributed computations, providing a less restrictive execution model compared to preceding frameworks, such as MapReduce [5]. As such, high performance is expected of the platform, and in order to achieve this, its SQL engine, Spark SQL, uses a highly efficient data-centric *code generation* approach for the execution of relational queries on big data, which was first introduced in HyPer [17]. A consequence of this approach is that the generated code is difficult to observe, and the different operators in the pipeline cannot easily be distinguished. Thus, although the Spark UI provides the user with detailed information on query execution, it does not currently provide a profiler that can report the execution time spent in these operators. As a result, it is difficult for a user to rewrite suboptimal SQL queries for better performance. In this thesis, we seek to create a novel approach of profiling generated code pipelines in SQL execution plans, that helps a user find the problematic operators in pipelines that were compiled to JVM bytecode.

Most SQL engines use the Volcano model [7] for the execution of queries. In this model, each operator acts as an iterator, and for each tuple requested, every operator in the query plan must be traversed via their iterator interfaces in order to obtain the next tuple from the data source. In order to assist users with diagnosing the cause of performance problems, many of these execution engines, such as those found in Oracle Database, MySQL, and PostgreSQL, provide the user with profiling tools that can identify the running cost of operators in a query plan. However, a problematic aspect of the Volcano model is that every operator incurs a necessary overhead introduced by the operator boundaries, requiring materialization of the data as tuples for every boundary. Spark uses the code generation approach to mitigate this, by fusing non-blocking, pipelining operators together, and using just-in-time (JIT) compilation to combine these pipelines into a single *for*-loop, thereby eliminating the materialization at operator boundaries in the pipeline, and allowing for rigorous compiler optimizations. Figure 1.1 shows an overview of this approach, and the challenges it poses for profiling. As of yet, none of the implementations of this approach provide a solution for profiling the component operators of such a fused pipeline.

Another challenge in building an accurate profiler for query plans is the definition and evaluation of accuracy. Therefore, in this thesis we seek to find such a definition for accuracy, as well

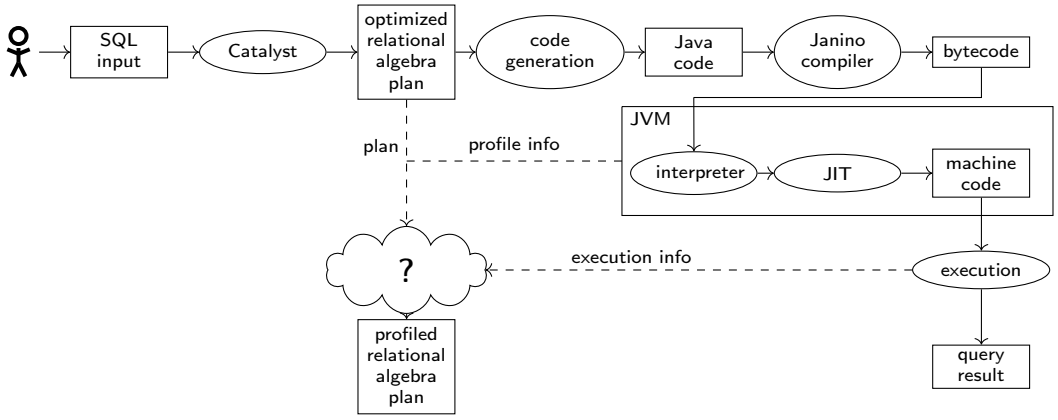


Figure 1.1: An overview of the profiling problem. The SQL input is passed through a number of optimization and compilation steps, making the collection of profiles a challenge.

as a framework for the evaluation of the accuracy of query plan profilers. In previous research, different metrics of accuracy for time-based profiling have been used, including correlation between successive runs [30], overlap with different profilers and the ability to correctly attribute the cost of artificial workloads [16].

Many profiling approaches for call graph edge profiling have a notion of a *perfect profile*, where all edges encountered in the execution of the program are accounted for [26]. However, within the scope of time-based profiling, there is always an uncertainty present in the profile caused by the execution environment, which includes effects of compiler optimizations, biases introduced by the CPU architecture, and I/O latency. Furthermore, the HotSpot JVM introduces various challenges in profiling, caused by the various non-deterministic stages of interpretation, optimization, and it's own JIT compilation of the program. These uncertainties make it impossible to obtain an exact runtime cost profile, and therefore, the result will always be an estimate. Moreover, besides the biases inherent to the runtime environment, every profiling method should be assumed to influence the running application. This *observer effect* can be caused by overhead introduced by sampling, interference with the JVM's profiler that drives optimization decisions, instrumentation that introduces bytecode fences or memory fences that influence JIT compilation and execution of machine code, as well as other factors that can be difficult, if not impossible, to correct.

In this thesis, we aim to find the most suitable approach for creating a profiling tool that can accurately profile partial query plans that are found as generated code pipelines in the executed query plan. In order to provide a frame of reference to help understand the requirements and performance of the profiler, we present a canonical Spark SQL query consisting of the operators Scan, Filter, Join and Aggregate (SFJA), which encompasses the operators that are most frequently found in SQL workloads, and which can co-exist in a single pipeline. The profiler will be designed to be able to profile this query. Different profiling methods will be explored, and in order to evaluate the accuracy of these profilers, artifically constructed SFJA queries will be used as an oracle for the correct runtime profile of a query. Finally, different methods will be explored for the use of Spark's task planning capabilities in order to sample and profile tasks, to obtain a method of confining the overhead, while still obtaining accurate profiles.

1.2 Problem statement

1.2.1 Research goals

The goal of this project is to find and evaluate a method of providing Spark users with a profiler that helps them to better understand what affects the performance of generated code pipelines in an optimized and compiled query plan, and how they might be able to optimize their queries for performance. Ideally, it should be able to be run in a production environment, such that a user can retrieve profiles of queries which have run in the past and have proven to be problematic. This section presents a number of criteria that such a tool should satisfy.

Firstly, the profiler should at the minimum be capable of aggregating data on the level of operators. This criterion derives from the goal of providing profiling information that is understandable to a user. Since a basic level of understanding of relational algebra is necessary for the effective use of Spark and Spark SQL, it is reasonable to assume that users of the platform have an understanding of the operators in relational algebra, such as the join, aggregation, project, and filter operations. However, most users cannot be assumed to have knowledge of the underlying implementation, and as such, these operators present the lowest level of understanding that can be expected of a user.

Secondly, the aggregations that are obtained by doing so must be trustworthy for the user. Therefore, the profiles generated by the method must be repeatable and accurate. Determining accuracy of a JVM profiler in the general case is difficult. Due to the observer effect, granularity limits, and CPU microarchitecture, it is difficult to determine where most of the time is spent in the execution of a program. In order to evaluate the accuracy of the Spark SQL profiler, an experimental setup is required where the profiler can be run in a controlled environment, such that a ground truth can be established.

1.2.2 Research questions

In this section, we present three research questions that are to be answered in this thesis. The first research question focuses on identifying existing or novel approaches of profiling within the JVM and Spark SQL. The second research question discusses the criteria for validity of these approaches, as well as the evaluation. The third and final question discusses opportunities presented by the integration of these approaches into Apache Spark.

RQ1. What approaches for obtaining performance profiles can be used for profiling the execution of generated code of an SFJA query pipeline?

The first step in obtaining a profile of SFJA queries is to obtain a profile of the physical execution, that is, a profile of the performance costs of the execution of bytecode that is generated from the SFJA query by Catalyst, within the JVM. In order to answer this question, two aspects have to be explored.

RQ1.1. What profiling methods exist for profiling Java programs?

The generated subroutines of an SFJA query are executed entirely within the JVM. Therefore, different existing approaches of generating profiles in the JVM profiling must be investigated, so an analysis can be done on the applicability of these profilers to the Spark SQL use case.

RQ1.2. What is the minimum granularity of the profiles provided by the identified approaches?

Finally, in order to be able to aggregate the results obtained by these profilers, the results must be fine-grained enough to allow for the attribution of the profiling costs to operators. This

granularity is determined by a number of different variables, such as biases due to the runtime environment, inhibitive overhead, and limitations of the debugging APIs.

RQ2. How can the accuracy of a profiler be defined within the scope of the SFJA queries?

In order to understand how well these profilers fulfill the criterion of high accuracy, a definition of accuracy of a JVM profiler must be found. The complex optimizations that are performed by the HotSpot architecture make it difficult to identify what the semantics are of the performance of a program. During the interpretation stage, the profile of a JVM application may look entirely different from the profile after optimization and JIT compilation. Furthermore, if a satisfactory definition of performance is found, it might still prove to be difficult to define accuracy in the general case of JVM programs. In previous literature, different metrics of accuracy have been used, such as overlap with other profilers, the correct identification of hot spots in the code, and the extent to which it can identify known workloads, however, these are problematic in the scope of the SFJA queries.

RQ2.1. How can an oracle be created that can establish an accurate baseline profile of controllable, synthetic SFJA queries?

As a consequence of the aforementioned difficulties with profiling in the JVM, establishing a ground truth for an accurate profiler is difficult, if not impossible to construct in the general case. However, the scope of this profiler is limited to generated code pipelines, hence a feasible approach might be to create an experimental setup where the performance cost of the individual operators in an SFJA query can be controlled, such that it can be used to evaluate the accuracy of the profiling methods that were identified in RQ1, in order to evaluate the accuracy of the method within the given scope.

RQ3. What is the impact of the integration of the identified approaches in Apache Spark?

It is desirable for the profiler to be easily integrated into the Apache Spark runtime environment. This brings with it a number of challenges, including limiting the overhead of the profiler in the execution environment, and the maintainability of the code base.

Chapter 2

Background

In this chapter, we discuss the background and related work. First we discuss the execution of SQL queries on distributed datasets in Apache Spark, and the problem this poses for profiling. Furthermore, we discuss existing work and industry tools of profiling methods, and how these might help us to build an accurate query pipeline profiler.

2.1 Spark SQL Execution Model

Spark SQL [3] is the SQL engine that is built into Spark, with the intention of providing a framework for the optimization and execution of highly efficient queries on relational datasets. This framework consists of a data structure for relational data, namely the *DataFrame* data structure, as well as the Catalyst query optimizer, which optimizes and compiles queries on these DataFrames. In this section, we will discuss how queries on dataframes are executed within Spark SQL.

2.1.1 Overview

DataFrames in Spark are inspired by the DataFrames provided by numerical computation frameworks such as *R* and *Pandas* [24, 15]. Analogous to the data frame data structure in Pandas, Spark uses a DataFrame interface as an abstraction over datasets that provides a column-oriented interface and schema describing the data in the dataset. Within Spark, this interface provides relational algebra operators, which allows a user of the framework to perform relational queries on distributed datasets.

An overview of the execution model that encompasses the operation and optimization of a SQL query in Apache Spark is presented in fig. 2.1. In order to compile SQL code and relational operations into executable Spark jobs, Spark relies on the Catalyst optimizer. The optimizer initially parses and resolves the SQL query using schemas from a catalog describing the relations known to Spark, to form a *logical plan* consisting of relational algebra operations operating on resolved. Initially, this logical plan is optimized and compiled according to the iterator-based *Volcano* model [7].

In the volcano model, each operator acts as an iterator that implements a *next* method that provides the next tuple in the operator's result set. During execution of a query within the Volcano model, the *next* iterator is called repeatedly on the root operator of the plan, until all results have been returned. If the operator has child operators, each *next* call will recurse down into the tree, calling the *next* method on each child operator to produce the next tuple. This is

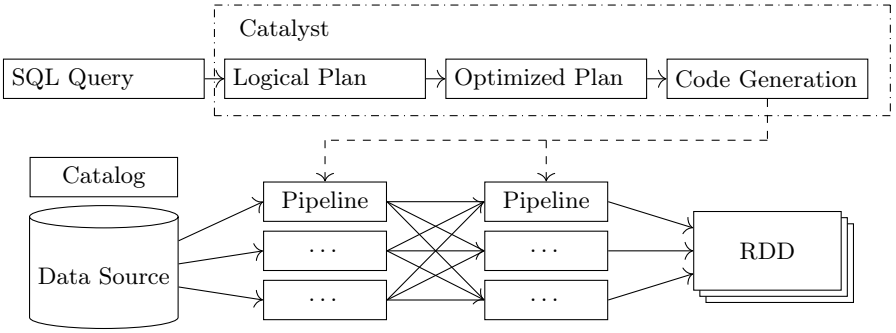


Figure 2.1: A high level overview of query execution in Spark SQL. The Catalyst optimizer compiles a SQL query into a query plan, and into separate pipelines using code generation.

called a *pull*-based approach, since data is pulled upwards through the tree from the leave nodes that typically represent data-access operators such as file scans.

Code generation A disadvantage of the volcano approach, as outlined above, is that in order to execute this query plan, the operator tree must be traversed for every tuple, incurring an overhead for every operator boundary each time the *next* method is called. In order to overcome this problem, Spark employs *generated code pipelines* to achieve Just-In-Time compilation.

Contributors to the overhead of the Volcano model have been identified as the manipulation of tuples that is required for each call because the operator is expected to produce tuples, as well as the overhead and lack of locality caused by the virtual *next* call [36], which requires indirection to be invoked. In order to overcome these problems, the HyPer SQL engine [17] introduced a data-centric *code generation* approach. This approach compiles the Volcano-model iterators into inlined generated code, producing a single *for*-loop that iterates over tuples.

To determine which operators in a query plan constitute a pipeline, this approach differentiates between *pipelining* operators which allow a tuple to be pushed directly to the parent operator, or *pipeline-breaking* operators such as read and write operators, aggregation or sorting operators. The latter require intermediate storage in memory, in which case materialization of the data as tuples for efficient access and data locality is necessary. As a result, these operators form the begin or the end of a pipeline.

In the original approach of HyPer, which was built using LLVM as compiler, this pipeline *dematerializes* the values in a tuple by loading the values from the tuple, and into registers in the LLVM virtual machine. In contrast to the Volcano model, this model is *push*-based, meaning that the tuples that are produced at the leaf nodes are pushed upwards through the tree one at a time, until the leaf nodes run out of tuples. This is achieved by inlining the code that is generated for each parent operator in the pipeline, and reusing the previously dematerialized values. Materialization of the separate values from the registers back into tuples is deferred to the top-most operator in the pipeline. As a result, each value in the output tuple only has been loaded and stored in a tuple once for a single pipeline. Furthermore, *next*-calls between the operators in the tuples are eliminated, allowing for a drastic reduction of overhead compared to the Volcano model.

Within the HyPer model, operators generate code fragments that either *produce* or *consume* a row. The *consume* fragment consumes one row as input from its child operator in the operator tree, and directly pushes the results upwards to the parent operator, without intermediate storage or materialization. Pipelining operators exclusively implement *consume* operations. For example,

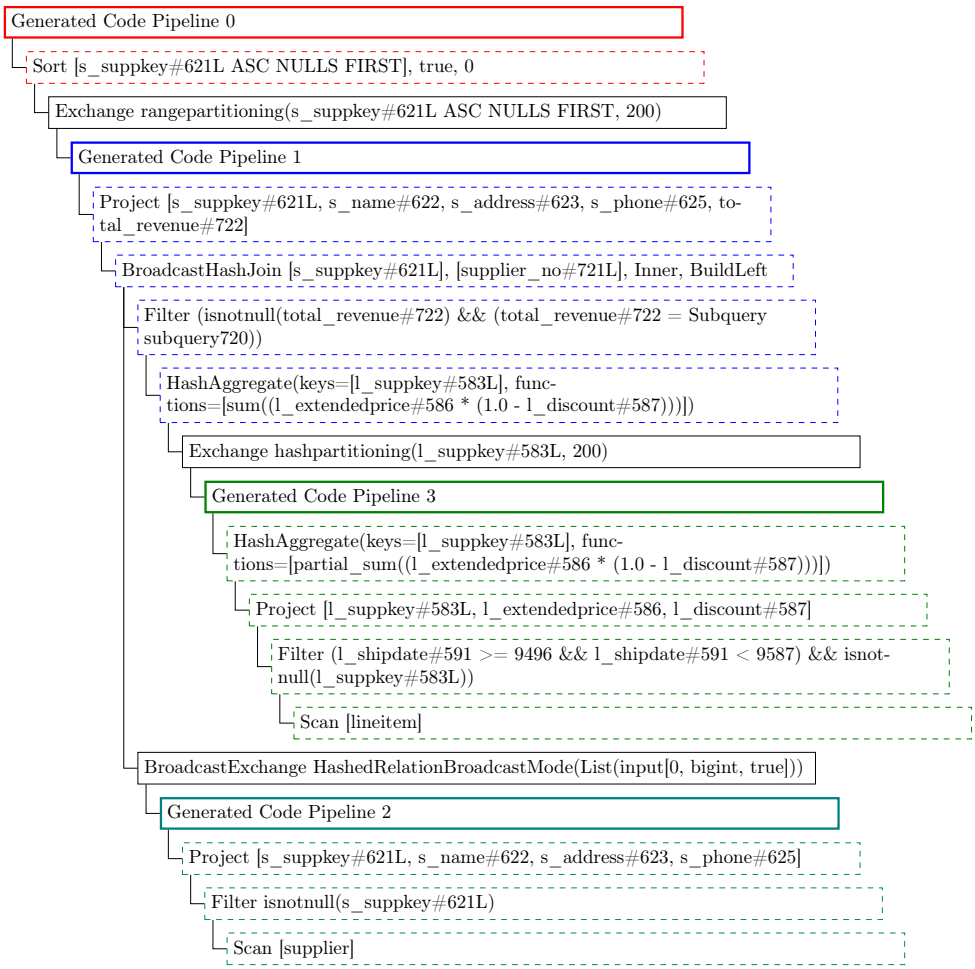


Figure 2.2: The physical plan for query Q15 in the TPC-H benchmark, illustrating the common occurrence of the operators *project*, *broadcast hash join*, *filter*, *hash aggregate* and *scan* in the compiled pipelines of an optimized SQL query.

Listing 2.1: A consume function generated for a broadcast hash join, a pipelining operator.

```
1 BroadcastHashJoin::consume(row) {
2   int join_key = row.getInt(0);
3   Row broadcast_row = broadcastRelation.getRowForKey(join_key)
4   if (broadcast_row != null) {
5     // parent call is inlined, whenever possible
6     parent.consume(row, broadcast_row);
7   }
8 }
```

a filter operator consumes a row from the child operator, checks it against the selected predicate, and pushes it upwards to the parent operator. Another example is the broadcast hash join operator, which searches its join key in a hash map to find the matching row, and pushes both the input row and the joined row into the parent operator.

Pipeline-breaking operators, such as scans or aggregations, do not fit this *consume* model and do not push a consumed row directly upwards. These operators implement the *produce* operation, which pushes new rows into the pipeline. If an operator requires it, both *consumes* and *produce* fragments can be implemented, for example in aggregations, where the entire pipeline is consumed before the partial aggregation results can be pushed to the output buffer.

To illustrate code generation, fig. 2.2 shows the query plan that is generated for query Q15 in the TPC-H benchmark [27], and listing 2.2 shows a simplified example of what the generated code for pipeline 3 of this query might look like. The pipeline consists of the operators *scan – filter – project – aggregate*, with global aggregation in Spark, where both *produce* and *consume* fragments are implemented by different operators. This is a simplified representation, the full generated class is over 600 lines long, and includes additional null-checking and aggregation logic, as well as the implementation of the scan iterator. A more complete example of generated code that was captured from an actual Spark SQL query can be found in appendix B.

The amount of code that is generated makes profiling a challenging problem. A single for-loop can span hundreds of lines of Java code, and this makes it difficult for a non-expert to identify the problem in a query. This is illustrated by fig. 2.3, showing the number of lines of code generated for a selection of queries in the TPC-H benchmark.

Within Apache Spark, the generated code execution approach is implemented as a buffered iterator. In the example in listing 2.2, the iterator is accessed via the *next* call, which tries to read from the output buffer, and if the buffer is empty, the buffer is filled in batches, by calling the *processNext* method in the iterator. This iterator loads a batch of tuples from the underlying iterator, which is implemented as the *scan* operation that provides the *produce* operation that iterates over the input relation, and subsequently passes these upwards through the operator tree, invoking the operators *filter*, *project*, and *hash aggregate*, where the calculated price is summed together.

In contrast to HyPer, the code generated by the operators is Java source code. As with the HyPer engine, to evaluate an operator, its input values must be loaded into a CPU register. In order to do so within Spark's code generation architecture the values are loaded into Java variables, and the register allocation is performed by the JVM. Spark will then try to split the operators into separate methods whenever possible, however, these methods can only be built if all necessary data is dematerialized into Java variables.

Listing 2.2: Simplified representation of a generated code pipeline for the SQL query `SELECT sum(l_extendedprice) FROM lineitem WHERE l_shipdate > date '1996-01-01'`

```
class GeneratedIterator extends BufferedRowIterator {
  double agg_accumulator_0;
  Iterator<Row> scan_iterator;
  RowWriter rowWriter;

  @Override
  public void init(Iterator[] inputs) {
    this.agg_accumulator_0 = 0;
    this.scan_iterator = (Iterator<Row>) inputs[0];
    this.rowWriter = new RowWriter(1, 0);
  }

  @Override
  protected void processNext() {
    // BEGIN PRODUCE HashAggregate
    this.agg_doAggregate();
    // output the result
    /// BEGIN CONSUME WholeStageCodegen
    this.rowWriter.reset();
    this.rowWriter.write(0, this.agg_accumulator_0);
    // append row to internal buffer
    this.append(rowWriter.getRow());
    // END CONSUME WholeStageCodegen
    // END PRODUCE HashAggregate
  }

  private void agg_doAggregate() {
    // initialize aggregation buffer
    // BEGIN PRODUCE Scan
    for (Row row : this.scan_iterator) {
      // BEGIN CONSUME Filter
      do { // short circuiting do block
        int value_l_shipdate = row.getInt(2);
        if (!(value_l_shipdate > 9496)) {
          continue; // short circuit the do block
        }
        // BEGIN CONSUME Project
        // BEGIN CONSUME HashAggregate
        double value_l_extendedprice = row.getDouble(3);
        double value_l_discount = row.getDouble(3);
        double value_3 = l_extendedprice * (1.0D - value_l_discount);
        this.agg_accumulator_0 += value_3;
        // END CONSUME HashAggregate
        // END CONSUME Project
        // END CONSUME Filter
      } while (false);
    }
    // END PRODUCE Scan
  }
}
```

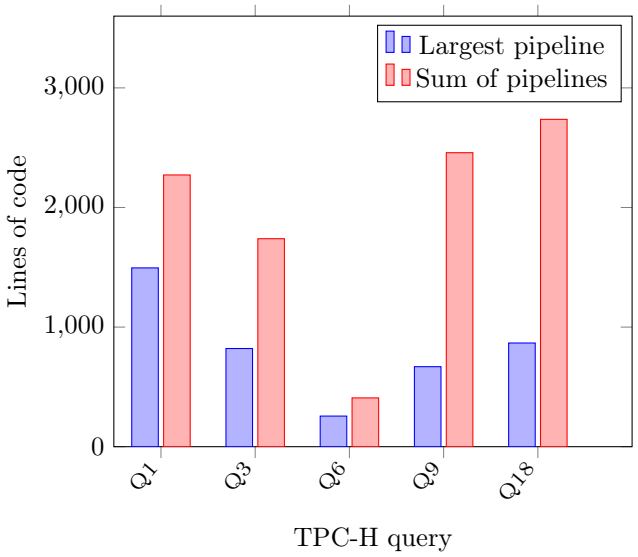


Figure 2.3: Lines of generated code for a selection of queries from the TPC-H benchmark

2.1.2 Query Execution

Spark is a framework for distributed execution. In order to achieve this, a Spark cluster consists of a driver node, which provides an interface to the user, and handles job execution on the cluster. The execution of the job is performed by executor nodes.

In order to distribute the a job over the executors, as well as to mitigate failures in job execution, Spark uses the *Resilient Distributed Dataset* (RDD) abstraction over data sets, which is similar to MapReduce [5]. This section describes the scheduling of SQL jobs operating on RDDs within Spark.

An RDD is a dataset consisting of a distributed set of partitions, that support two types of operations, namely *transformations* and *actions* [33]. A transformation is an operation that yields a new RDD with the transformed data, and an action is an operation that returns data from the RDD to the application. A transformation is executed lazily, meaning that only once an action is called on an RDD, the transformations leading to that RDD are executed. As such, each RDD keeps track of a DAG consisting of RDDs as nodes, and transformations as edges, that are required to obtain that RDD. This graph is called its *lineage graph*. If a part of the job fails, and partitions of an RDD are lost, these lost partitions can be recovered from its source RDDs by traversing the lineage graph.

A transformation on an RDD can be *narrow*, such that only data from one partition is used, or it can be *wide*, meaning that data from multiple partitions is required to perform the transformation. In the latter case, a shuffle occurs, in which data is exchanged between different operators. An example of a narrow transformation is a simple map, whereas a reduce operation is an example of a wide operator.

The way SQL execution ties into this, is through the implementation of DataFrames as an abstraction over RDDs, and each execution plan of a SQL query is executed as a DAG of RDDs and transformations. Within this execution model, the generated Java code of a query pipeline acts as a narrow transformation on the RDD, mapping one RDD partition to another, whereas pipeline-breaking operators, such as sorts or an aggregations, typically result in a wide transformation.

This implementation of distributed computing has consequences for the execution of Spark SQL queries. The generated code pipelines allow for efficient narrow transformations on partitions, whereas wide transformations are required for many pipeline-breaking operators such as sort, large join operations, or aggregations.

In order to execute a job that is invoked by the action on a Spark cluster, Spark will split the execution up into separate tasks of that are applied to the separate partitions in the RDD. The tasks are then scheduled on different nodes in the cluster using a DAG scheduling algorithm such that each task is scheduled only if the required partitions are available. Furthermore, the scheduler is capable of speculative execution of tasks. Within cloud platforms, virtual instances of Spark worker nodes in a cluster are often observed to be heterogeneous [34], which can negatively impact performance. In order to mitigate performance issues due to this heterogeneity, speculative execution is used in order to replicate the same task across different virtual machines in order to find a machine where this slow-down does not occur. This mechanism is interesting from a profiling point of view, since profiling is also expected to incur an overhead. Task duplication might provide a means of confining this overhead.

2.1.3 Java Virtual Machine

Apache Spark is designed to run in the HotSpot Java Virtual Machine (JVM). As such, it is important to take into account the consequences of execution within the JVM on the application and the profiling. This subsection discusses some relevant aspects of the HotSpot JVM.

Within the HotSpot JVM, a method can be executed in two ways, namely interpreted, and Just-In-Time (JIT) compiled. The JVM achieves this through a strategy called *tiered compilation* [21]. This compilation consists of one level of interpretation and four levels of compilation. The four compilation levels each use one of the two different profile-guided optimizing compilers C1 and C2, which compile the bytecode of a method into low-level machine code for the target architecture and operating system. Each compiler has a separate queue, to which methods that are to be compiled are submitted. Each compiler will take compilation tasks from their respective queues and perform these tasks continuously throughout the execution of a Java program. C1 applies relatively simple optimizations, and compiles faster, whereas C2 applies more advanced optimizations, at the cost of slower compilation times. The following levels of compilation are available [28]:

Level 0	Interpreter, with full profiling.
Level 1	C1 compiler, fully optimized, without profiling. Used for trivial methods.
Level 2	C1 compiler with invocation and loop back-edge counters. Fast execution, and fast compile times.
Level 3	C1 compiler with full profiling. The generated machine code executes slower than level 2, but collects more detailed data, including data on branches taken, the types of the object receiving the method call, and array accesses.
Level 4	C2 compilation with full profile-guided optimization. Faster than level 2, but compilation is slow.

Initially, bytecode will be run in interpreted mode at level 0. In this mode, each bytecode instruction is interpreted and executed separately, and the JVM is guaranteed to be in a consistent

state before and after each instruction. During execution, HotSpot collects runtime information about the executed code, and if a method is observed to be called often, it will submit the method to the C1 compilation queue. Initially, all methods are compiled by the C1 compiler, using the level 3 strategy, collecting a runtime profile. Once sufficient profiling information is collected, the profile and method are submitted to the C2 compiler in level 4, which applies optimizations based on this profile, to obtain a highly optimized method. For each compilation step, the compiled code is placed in a region of memory called the *code cache*. When a compiled method is no longer used, it can be removed from the cache. If the C2 compilation queue is too long, level 2 compilation is used as an intermediate step before compiling to level 3, to avoid spending a long time executing the slower code generated by level 3, while waiting for level 4 compilation to complete.

The JIT compilation is optimistic, meaning that the compiler makes assumptions to unroll loops and branches based on the collected profile. If these assumptions do not hold true and an uncompiled path is followed, an uncommon path trap is triggered, causing the JVM to fall back to interpretation at level 0, to follow the alternative code path [22].

With respect to profiling, this compilation process poses multiple challenges. Performance of the resulting code can differ drastically between different invocations of the JVM JIT-compiler, and code interpretation might take up a significant amount of time as well. In order to formulate a definition of accuracy of a profiler, these aspects should be taken into account. Furthermore, another complication introduced by the JIT compiler is caused by recompilation of methods, that makes it difficult to understand the relation between executed machine code and the methods that produced them. In particular, this can cause problems for profilers which rely on low-level call stack sampling and make the assumption that this compiled code is static. Furthermore, this poses a problem for long running programs that may load and unload methods on the fly, such as Apache Spark, since profiling information collected on a region of machine code may be collected on a method that has since been overwritten by a different compiled method.

2.2 Profiling Methods

In this section, we describe various methods of profiling that are used in industry or were introduced in previous literature. We classify them in two categories: *sampling profilers*, which stop a thread to take samples of the current execution state of that thread, in order to obtain a profile, and *instrumenting profilers*, which modify the running code to include performance statistics reporting code at runtime.

2.2.1 Performance Monitoring Units

To provide the tools necessary for introspection into the performance of a running machine, modern CPUs are equipped with Performance Monitoring Units (PMUs), which provide access to various performance monitoring counters [31][1] that provide insight into the performance aspects of a running program. Events that are recorded by the CPU include, but are not limited to, clock cycles, retired instructions, and the number of cache misses for various cache levels. These counters can be read during the runtime of a process, and moreover, these counters can be programmatically configured to interrupt the process once a programmed overflow value has been reached.

Some features of these PMUs offer low level hardware access to the programmer, thus, secure access to these counters has to be regulated via the operating system. Windows, MacOS and Linux provide the APIs *dtrace*, *perf_ events* and Windows counters, respectively, that allow the user access to these counters.

PMU events	branch-instructions, branch-misses, bus-cycles, cache-misses, cache-references, cpu-cycles, instructions, ref-cycles
Software events	alignment-faults, bpf-output, context-switches, cpu-clock, cpu-migrations, emulation-faults, major-faults, minor-faults, page-faults, task-clock

Table 2.1: Software events and generic hardware events available to *perf*. The names shown are generic names, as defined by the Linux kernel, of hardware-specific PMU events.

Many popular profiling tools, such as those based on the *perf_ events* interface, perf [14] and OProfile [13], as well as other profilers such as Intel VTune [10], provide user interfaces to these performance counters, allowing the user to profile an application and to provide insight into various aspects of the performance of a specific running program, or of the system as a whole. Table 2.1 shows an overview of PMU events that the Linux kernel exposes by default for *perf*.

Since Spark is a platform for distributed computing, it is often run in cloud environments, and clusters are often run on virtual machines in a multi-tenant environment. Allowing virtual machines to access the PMU leaves other virtual machines vulnerable to side channel attacks [35]. To prevent this, on many cloud computing platforms such as Amazon AWS and Microsoft Azure, the hypervisor forbids access to the PMU, and only on single-tenant instances, such as the dedicated hosts provided by Amazon AWS, the hypervisor may allow usage of hardware PMUs [9].

In order to still be able to provide performance counters when hardware PMUs are unavailable, the *perf_ events* interface provides a limited set of software-based event counters, shown in table 2.1, that provide event counts generated by the Linux kernel.

2.2.2 Instrumenting profilers

A common method of obtaining profiles from applications at runtime is *instrumentation*, the modification of running code, in order to inject specific instructions that allow for or perform the collection of profile information.

Instrumentation is performed in the HotSpot JVM itself, as part of its tiered compilation [21]. In compilation levels 2 and 3, the JVM compiles back-edge, branch, and invocation counters in the compiled code, which allows the JVM to apply optimizing strategies based on usage, and perform optimistic optimizations based on the collected branches. This does not make use of hardware performance counters, however, and only the edge counters are taken into consideration for the optimization.

NanoLog [32] is another approach for instrumentation is a high-performance computing oriented logging tool that is designed to provide logging at high resolution in native applications, capturing the current time in CPU execution at nanosecond resolution. It does so by leveraging compile-time optimization through the use of C++ macros, and writing compressed files to minimize I/O overhead. Furthermore, it uses thread-local allocation buffers to queue messages for writing. It achieves a throughput up to 80 million messages per second, which is an order of magnitude faster than comparable tools such as Log4J and the Java and C++ standard library standard output implementations.

Instrumentation in the JVM The Java-specification includes an optional interface to perform instrumentation and introspection on running JVM applications, called the JVM Tool Interface (JVMTI). The JVMTI provides various event handlers, including the handler named

ClassFileLoadHook [20], that allows an agent to read the bytecode of the class and modify or substitute it before its contents are loaded by the JVM. This is commonly used for the instrumentation of methods in the class, for example to inject performance probes.

An approach to profiling applications in the JVM using the JVMTI is dynamic instrumentation of bytecode, as used in the JP2 profiler [26]. Through dynamic instrumentation of bytecode, and inserting performance counters in the running code, JP2 can provide full coverage over a region of interest. Although the instrumented code has a significant overhead, the instrumentation is done dynamically at runtime, and as a result, the overhead can be limited by disabling the instrumentation at intervals. However, this is expected to affect runtime behavior, due to the JVM JIT compiler being forced to recompile the method after the instrumented bytecode is modified.

A logging approach such as this is a feasible profiling solutions for smaller applications, or for monitoring at a greater granularity. With a big data platform such as Spark, however, generating many log messages per row would result in an output dataset that is many times greater than the number of input rows, producing an infeasible amount of data.

As an alternative to tuple-at-a-time approaches like the Volcano model and code generation, other database engines use column-at-a-time execution approaches [4, 12, 36]. These operate by performing operations one at a time on a column-by-column basis. This has the advantage that the execution of an operator is a standalone subroutine that does not call other operators. As a result, profiling of this approach is trivial, and the time spent inside an operator can simply be measured by reading a performance counter before, and after execution of the operator, at minimal cost. As such, the instrumentation approaches described above would be highly accurate and feasible.

2.2.3 Sampling profilers

Industry sampling profilers operate by interrupting the execution of a process at regular intervals, and capturing information of the current call stack of that process.

If CPU usage is to be measured, the sampling interval is taken as a period of CPU clock cycles. However, some profilers, such as those built on the *perf_ event* [14] or *DTrace* [2] interfaces, also allow for other PMCs to be used, so sampling might be used to find performance bottlenecks caused by branch misses or other hardware events.

An analysis by Mytkowicz et al. [16] shows that profiling at fixed intervals can result in biased results, which is caused by aliasing between the sampling period and periodicity in the executed code. To prevent this from affecting the result, they recommend the sampling period to be randomized between samples. In practice this can be challenging, because many APIs such as DTrace and *perf_ event* do not natively provide such functionality.

In a comparison between different approaches by Nisbet et al. [18], another useful tool for sampling native stack frames was found to be the *extended Berkeley Packet Filter* (eBPF) based tool *bcc*. This also allows for profiling of Java applications with the use of *perf-map-agent*. eBPF allows for programmable tracing and sampling at the kernel level, and allows profiling tools to trace or sample advanced performance events from within the kernel. However, this requires a recent Linux kernel, 4.4 or later, and is unavailable in many cloud environments. Like *perf*, it does require *perf-map-agent* to run.

A different approach of sampling performance monitoring tools is introduced by the tool Shim [32]. In this approach, observer threads are used to autonomously collect CPU performance counters on a target thread at a very low overhead, and at a fine resolution, allowing for sampling periods as small as 15 CPU cycles. In this approach, an observer thread is used to capture these performance counters, thereby delegating the I/O required for profiling to a thread separate

of the executing thread. This approach is targeted at a lower level of performance monitoring than what is needed for SQL profiling, and has the goal of finding the exact cause of low-level performance events such as cache or branch misses. However, the idea of using a listener thread to obtain profiles for the running thread is applicable to the SQL pipeline use case.

Sampling in the JVM Within the JVM, various approaches for sampling exist. Many JVM implementations provide the JVM Tool Interface (JVMTI) [19]. This API exposes various functions that allow for introspection into JVM execution. Many commercial profilers, such as *YourKit* and *xprof* use this API to obtain call graph profiles [16], where the *GetAllStackTraces* method is used to obtain a call stack of of every Java thread currently running in the JVM.

This method, however, suffers from *safe-point bias*. Within the HotSpot JVM, JIT compiled methods may bring the thread temporarily to an inconsistent JVM state, thus the behavior of operations on the thread's state are undefined, and may yield invalid results, or even cause fatal errors. In order to perform garbage collection, the thread must be brought to a consistent state. Therefore, a JVM thread can be requested to yield execution at the first point where the thread can be guaranteed to be in a consistent state, a so-called *safe point*. These safe points only occur between interpreted bytecode instructions, during the execution of native methods, loop back edges, and on entry and exit of JIT compiled, non-inlined methods. Since the JVM performs aggressive inlining, small, hot methods are almost guaranteed to be inlined, and thus they are underestimated in the resulting profile. In an analysis comparing various implementations of this approach, these implementations are shown to often underestimate the hottest methods, and attribute performance costs to methods that are not part of the call stack where the CPU time is actually spent [16].

A profiling method that avoids this bias samples native stack traces from the JVM JIT compiled code, which can be done safely without bringing the thread to a safe point. The native stack frames can then be resolved to JVM stack frames using the JVMTI interface. A popular tool that uses this approach is *perf-map-agent* [25], which uses Linux' *perf* tool as a profiler for native stack traces, providing a very low overhead and high precision. A disadvantage of this method is that interpreted JVM frames do not produce native stack frames, and therefore cannot be profiled by these tools.

Other profiling tools, such as *honest-profiler* [29] use the HotSpot JVM-specific API handle `AsyncGetCallTrace` [21], which is able to retrieve a call trace from a thread, without having to wait for the thread to enter a safe point. It does this by allowing the retrieval of the call trace to fail if the state of the thread does not allow a call stack to be retrieved, and as such, not all samples are successful. Furthermore, this profile does not obtain call stack samples for native methods.

The tool *async-profiler* [23] uses an approach that combines the above two approaches of *honest-profiler* and *perf-map-agent*, allowing it to obtain an accurate profile of JVM-specific methods as well as native methods, without requiring the use of a a JIT address mapping. Furthermore, async-profiler uses thread-local buffers for data locality and to avoid synchronization, allowing the profiler to collect samples at a low overhead and without interfering with other threads, yet yielding very accurate profiles at the granularity of Java methods. Furthermore, it includes a powerful visualization tool called *flame graphs*, which were originally developed by Brendan Gregg [8]. Since *async-profiler* does still rely on the `AsyncGetCallTrace` method to obtain stack traces, not all samples are guaranteed to produce a complete call stack.

A similar profiling approach is provided by a commercial tool in Oracle JDK 8, called Java Flight Recorder. This tool uses a different approach of capturing call stacks, by suspending the thread and providing call stack information of the suspended thread. However, this is not available for the open source OpenJDK 8 implementation that Spark is built for and that is used

by most users of the platform.

2.3 Conclusion

The code generation approach as outlined in section 2.1 is what sparks the question of how to provide accurate profiles for these pipelines. In order to provide a detailed profile for such fine-grained code, a profiling method is necessary that can provide profiles with high granularity.

The profiling methods that we identified in previous research and in industry tools are not sufficient to provide such high granularity profiles for JIT-compiled Java applications. Existing tools provide either results with a coarse granularity that is not sufficient for highly detailed profiling, or they are capable of providing high-granularity results, but do not provide the necessary information to aggregate these results at a detailed enough level. Past research does, however, provide approaches that allow us to collect fine-grained profiles for native execution, and although we haven't been able to find a ready-made tool that uses this approach, nor an evaluation of the accuracy of this approach, the JVMTI provides us with the JVM introspection necessary to be able to aggregate these native profiles at a granularity as fine as individual JVM bytecode instructions or individual lines of Java code.

Chapter 3

Design considerations

In this chapter, we will discuss the different approaches that we explored, and the design decisions that were made for these approaches.

3.1 Canonical query

In order to better understand the requirements of a profiling approach for Spark SQL query pipelines, a canonical query was designed that can stand as a representative model for other queries, with the goal of providing a succinct, but representative analysis of the performance of the profiler. The purpose is to operate within the scope of a compiled pipeline, and as such, it is desirable that the work of the canonical query is dominated by the work done in a single pipeline. Furthermore, the query should encompass relational operators that typically occur in a single generated code pipeline, and the generated code must follow patterns that are commonly found in executed code. Furthermore, the pipeline must be representative of the memory footprint of a typical query, and must span sufficient CPU registers to obtain representative results.

The Catalyst optimizer is a rule based engine, and as a result of commonly applied rules, compiled query plans for different queries are often found to be following the same patterns. Ignoring the *project* operator, which – with an exception for rare cases where the project applies a non-deterministic function – is optimized away by code generation, one such pattern that is regularly found in queries on relational data is the pattern *Scan–Filter–Join–Aggregate*. This pattern was chosen to be the dominating pattern of the canonical query, since it encompasses the most common operators. Thus, we define the the canonical SFJA query as the SQL query shown in listing 3.1.

This query is semantically meaningless, however, it was chosen to fulfill the conditions for Spark to optimize it into a SFJA pipeline. The GROUP BY clause was chosen to include keys from both sides of the join, in order to prevent the aggregation from being pushed down into a subtree of the join operator.

The full query plan for the SFJA query is shown in Figure 3.1. It performs as scan over a relation A, applies a filter with a given predicate, which is usually pushed down from above the join, and joins it with a relation B, before applying a partial aggregation within the partition. This pattern is often compiled into a single generated code pipeline.

In order for this pattern to occur as a single pipeline, the relation B must be small, such that Catalyst can apply the *broadcast hash-join* strategy, broadcasting the entire relation B over the partitions in A. This way, the join operation can be performed locally, on the partitions

Listing 3.1: The definition of the SFJA query

```
SELECT
   (l_suppkey * 100) + s_nationkey,
  AVG(l_extendedprice),
  SUM(l_quantity),
  AVG(l_extendedprice),
  AVG(l_discount)
FROM lineitem
JOIN (
    SELECT * FROM supplier
    ORDER BY s_acctbal DESC
    LIMIT 10000
  ) ON l_suppkey = s_suppkey
WHERE l_commitdate > $MIN_COMMIT_DATE
GROUP BY (l_suppkey * 100) + s_nationkey
```

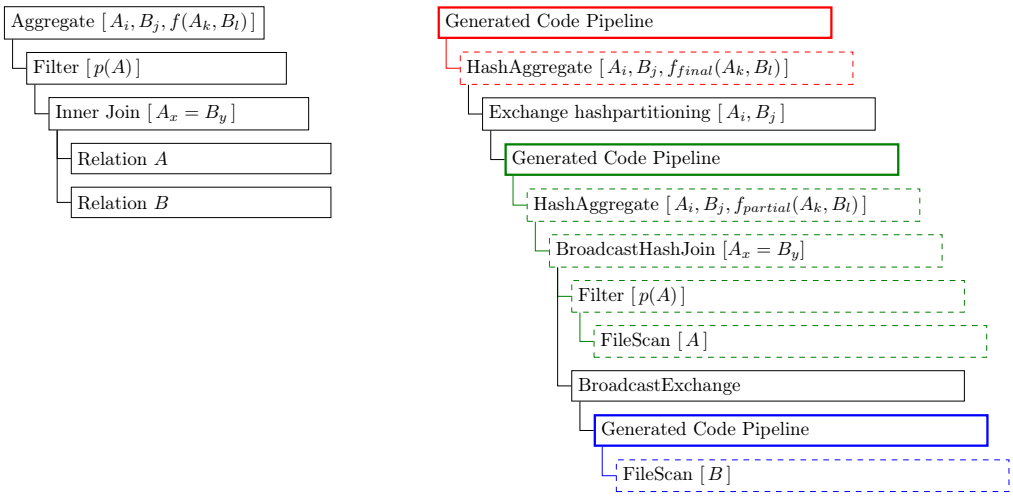


Figure 3.1: The logical plan for the canonical Scan–Filter–Join–Aggregate query, and its corresponding physical plan after optimization, with the generated code pipelines and their component operators highlighted. The subtree highlighted in green shows how the SFJA query manifests itself as a single generated stage.

of A, without exchanges within the pipeline of A, and without materialization of rows in A. This is a common operation that typically occurs when joining tables of different cardinalities. Since queries commonly compile into a query plan with this pattern, the pattern is seen often in executed queries, and a profiler should be capable of accurately profiling this pattern. Therefore, this pattern is chosen as our canonical pattern.

The filter and join operators do contain some intrinsic selectivity. As a result, the filter operator is the costliest operator in the baseline, followed by the join, and finally the aggregation.

3.2 Profiling methods

In chapter 1, we have explored a number of approaches that might be used to obtain fine-grained profiles in the Java Virtual Machine. In previous literature, as well as in industry, numerous profiling approaches have been introduced, however, none of these approaches are expected to be sufficiently applicable for distributed compiled query plans.

Within the use case of SFJA queries, these approaches lack the granularity or means of aggregation that are needed to apply them to these query plans. Therefore, in this section we explore four new approaches that extend on the ones identified, and identify what changes to Apache Spark or the respective profiling approaches are necessary to profile these pipelines. The approaches that we will discuss are the following:

Timing and tracing instrumentation (TTI)	By collecting timing data at fixed points in the code, information can be gathered about the timing of specific areas of interest.
Instrumentation-aided sampling (IAS)	This approach introduces instrumentation to keep a global variable in memory that lets a profiler collect the current operator that is being executed.
Method-level stack sampling (MLSS)	Existing profilers are capable of sampling at the level of Java methods. By splitting the operators, which are normally inlined in the generated code, into separate methods, these existing profilers might be used to collect fine-grained profiles.
Line-level stack sampling (LLSS)	Using the JVMTI and a native profiler such as *perf*, a fine-grained profile can be collected that can be aggregated at the level of lines of code. By mapping this back to the source code of a generated pipeline,

3.2.1 TTI

A common approach of profiling code at runtime is the collection of *tracing and timing instrumentation* (TTI) at fixed points in the code. One way this can be achieved is by instrumentation of the running code at runtime [26, 11], in order to execute code that logs events at certain points in the execution. Within the JVM, a typical method of achieving this is by modifying the bytecode directly when a Java class is loaded by the virtual machine, but other approaches might include modifying the generated machine code, or direct manipulation of the source code.

Listing 3.2: The main aggregation loop of the pipeline shown in listing 2.2, instrumented with `System.nanoTime` collection.

```
1  private void agg_doAggregate() {
2    long scan_startTime = System.nanoTime();
3    for (Row row : this.scan_iterator) {
4      long filter_startTime = System.nanoTime();
5      do {
6        int value_o_orderdate = row.getInt(2);
7        if (!(value_o_orderdate > 10227)) {
8          continue; // short circuit the do block
9        }
10       long project_startTime = System.nanoTime();
11       long agg_startTime = System.nanoTime();
12       double value_o_totalprice = row.getDouble(3);
13       this.agg_accumulator_0 += value_o_totalprice;
14       agg_totalTime += agg_startTime - System.nanoTime();
15       project_totalTime += project_startTime - System.nanoTime();
16     } while(false);
17     filter_totalTime += filter_startTime - System.nanoTime();
18   }
19   scan_totalTime += System.nanoTime();
20 }
```

The advantage of this approach is that it's not only suitable for profiling method call graphs, but if required, it can also generate structured profiles when structure of the executed machine code does not provide a one-to-one match to the structure of the source that generated this code. In operator pipelines, this is the case, since multiple operators in the generated code are often inlined into a single method. Instrumentation allows for the definition of regions of interest within the generated code pipeline.

As a naive instrumentation approach, the generated code could be instrumented such that for each row that passes through the stage, events are registered each time an operator is doing some work. By taking the `System.nanoTime` timestamp on the entry and exit of the region of code generated for that operator, and summing this throughout the execution of the code, a profile can be built. Within the JVM, the collected timestamps can then be aggregated to provide a wall-clock profile for each operator. If the proper corrections for the overhead of these calls are applied, and the resolution of the system clock is precise enough, this might be able to yield accurate profiles. However, since the CPU might execute parts of this call during time at which it is otherwise stalled, the actual cost of this system call might not be uniform across the workload, and biases might be introduced. Furthermore, this approach is likely to have a high overhead, and moreover, it might inhibit code optimizations, which further increase the overhead and reduce the accuracy of the profile.

To analyse what might be expected of such a solution, a small benchmark was run, as shown in listing 3.3 aggregating `System.nanoTime` calls in a simple loop. This benchmark shows that the `System.nanoTime()` method carries an overhead of approximately 15 nanoseconds on an x86 Linux system[1], and has a granularity of approximately 15 nanoseconds. This is an order of magnitude greater than cost of applying a simple selection operation on a row, which might take less than a nanosecond to complete. Another problem introduced by this method is the need to swap additional data into registers for each operation, introducing additional overhead. Therefore this approach is not expected to result in accurate profiles, and it is expected to

[1]Linux 5.0.0-29-generic, Ubuntu 18.04, OpenJDK 1.8.0_222. CPU: Intel Core i7 4790X (Haswell)

Listing 3.3: A benchmark of `System.nanoTime()` on a linux system

```
class Benchmark {
    public static void main(String[] args){
        long N = Long.parseLong(args[0]);
        long sum = 0;
        long prev = System.nanoTime();
        for(long i = 0; i < N; i++){
            long t = System.nanoTime();
            sum += t - prev;
            prev = t;
        }
        System.out.println(String.format("%f ns.", sum / (double)N);
    }
}
```

```
$ time java Benchmark 1000000000
15.179947586 ns
java Benchmark 1000000000
 15,20s user
 0,02s system
 100% cpu
 15,221 total
```

introduce a large overhead.

Listing 3.2 shows how this tracing method manifests itself in a generated pipeline. In order to run this, many variables must be declared that aggregate the times spent in the operator, occupying registers, and complicating register allocation for the JVM.

3.2.2 IAS

Another approach that was explored, is *instrumentation-aided sampling* (IAS). This approach keeps a variable in memory that indicates the current location in the operator tree, throughout the execution of the pipeline. In order to do so, some instrumentation is necessary in order to keep that variable in memory, and to keep it updated on every transition between operators. By using sampling, a profile can then be built using the observed values of the collected method.

A challenge with this approach is that this variable must be available asynchronously, throughout the execution of the Java code, and, because of the fine granularity of the code, it must be kept synchronized in memory, which requires memory barriers. The instrumentation of this approach is illustrated in listing 3.4. Here, a *DirectByteBuffer* is used to provide a region of memory that is not managed by the JVM's garbage collector, and is guaranteed to stay in one place in memory. The `synchronized` blocks are used to provide the appropriate load barriers. Furthermore, the variable is expected to be a direct match to the corresponding bytecode instruction, and if we can find an approach to match the instruction pointer to the current bytecode instruction, keeping this variable in memory is redundant.

3.2.3 MLSS

The first stack sampling approach that was explored is *method-level stack sampling* (MLSS). This is a relatively simple approach of profiling, that is used in various industry tools, either using

Listing 3.4: The main aggregation loop of the pipeline shown in listing 2.2, instrumented with the IAS profiler instructions.

```
1  final ByteBuffer byteBuffer;
2
3  private void agg_doAggregate() {
4    synchronized(byteBuffer) { byteBuffer.putInt(0) };
5    for (Row row : this.scan_iterator) {
6      synchronized(byteBuffer) { byteBuffer.putInt(1) };
7      do {
8        int value_o_orderdate = row.getInt(2);
9        if (!(value_o_orderdate > 10227)) {
10         continue; // short circuit the do block
11       }
12       synchronized(byteBuffer) { byteBuffer.putInt(2) };
13       synchronized(byteBuffer) { byteBuffer.putInt(3) };
14       double value_o_totalprice = row.getDouble(3);
15       this.agg_accumulator_0 += value_o_totalprice;
16       synchronized(byteBuffer) { byteBuffer.putInt(2) };
17       synchronized(byteBuffer) { byteBuffer.putInt(1) };
18     } while(false);
19     synchronized(byteBuffer) { byteBuffer.putInt(0) };
20   }
21   scan_totalTime += System.nanoTime();
22 }
```

the JVMTI interface [6], the `AsyncGetCallTrace` method [23, 29], a profiler can be created that collects accurate call stack profiles. However, since code generation produces large methods with inlined operators, these call stacks alone are not informative with regards to the performance of the separate operators in the pipeline. In order to produce more informative results, we can leverage a mechanism that Spark uses to split operators into methods, and thereby improve the usability of these stacks.

In order to prevent excessively straining the C1 and C2 compilers, the HotSpot JVM enforces a limit of 8 kilobytes of bytecode on the compilation of methods. This entails that methods that exceed this limit are never considered for optimization, compilation or inlining. In normal code bases this is rarely a problem, however when code generation is used, such huge methods may occasionally present themselves in generated code.

To prevent this optimization limit from hindering query execution, Catalyst tries to compile the operator's consume operation as a separate Java method, rather than inlining the entire pipeline into a single method. Thereby it limits the size of the method, and allows the JVM to compile it. However, because of architectural choices within Spark, this can only been done if the condition is met that all input variables to the operator have been materialized before the operator is invoked.

Java will be able to inline these methods with ease as long as no limits are exceeded, so the overhead of these methods is expected to be limited. However, a larger overhead is expected to be caused by a constraint of Spark's implementation of these operator methods. In order to construct a new method for a simple operator, Spark requires all input parameters to be dematerialized and loaded into local variables. Under normal operation, Spark can avoid materializing many columns due to short-circuiting operators such as *filter* or *join*. When early dematerialization is applied, however, this short circuiting becomes less effective.

Listing 3.2 shows an example of the simple pipeline of listing 2.2 with split operators. By forcing dematerialization of all input columns for the operator, each operator can be split into

Listing 3.5: The main aggregation loop of the pipeline shown in listing 2.2, with the operators split into methods for MLSS. Notice that the *o_totalprice* column read is redundant when the filter rejects the row.

```
1  private void agg_doAggregate() {
2      scan_doProduce();
3  }
4
5  private void scan_doProduce() {
6    for (Row row : this.scan_iterator) {
7      double value_o_totalprice = row.getDouble(0);
8      int value_o_orderdate = row.getInt(1);
9      filter_doConsume(value_o_totalprice, value_o_orderdate);
10   }
11 }
12
13 private void filter_doConsume(double o_totalprice, int o_orderdate) {
14   do {
15     if (!(o_totalprice > 10227)) {
16         continue; // short circuit the do block
17     }
18     project_doConsume(o_totalprice, o_orderdate);
19   } while(false);
20 }
21
22 private void project_doConsume(double o_totalprice, int o_orderdate) {
23   agg_doConsume(o_totalprice);
24 }
25
26 private void agg_doConsume(double o_totalprice) {
27   this.agg_accumulator += o_totalprice;
28 }
```

separate methods, allowing existing Java sampling profilers to obtain the operator hierarchy as a stack trace. Note that these input columns, such as *o_orderdate* in the example, are not always dematerialized in the original pipeline.

Another disadvantage of this method is that the dematerialization required for the invocation of the *consume* method of the parent operator is performed in the child operator's *produce* code, and therefore is misattributed, since this work is logically performed by its parent operator. As a result, we expect an overestimation of the cost of the topmost *produce*-section in the call stack.

3.2.4 LLSS

An alternative sampling approach that we have explored is *line-level stack sampling*. We can achieve this using native instruction-level sampling to obtain a fine-grained profile of the machine code. By using events exposed via the JVMTI, metadata on method loading and compilation can be collected to obtain a mapping from native addresses of JIT-compiled methods to lines of Java code. By sampling native stacks using Linux' *perf* tool for Java, fine-grained profiles can be obtained on the level of instruction addresses. The collected profile can then be aggregated using the collected lines of code that generated the instructions, and subsequently at the level of the corresponding SQL operator that produced it. This provides a means of aggregation of instruction pointer samples on the operator level.

A drawback that this approach presents is that in order to gather call stacks using instruction

addresses, a frame pointer needs to be available for each stack frame, that's pointing towards the previous stack frame, such that the profiling agent can walk through the call stack structure by following the frame pointers. Walking through this stack is a necessity to obtain accurate profiles, even with inlining enabled, because complex functions external to the generated code might not be inlined, and these can be especially costly to execute, and therefore are expected to have a large impact on the runtime profile. Only by allowing a profiler to walk through the stack, these functions can be traced back to the operator that invoked it.

Conventionally on x86_64 platforms, the *RSP* general-purpose register is used for the purpose of keeping the stack structure. However, by default, the HotSpot JVM performs optimizations that allow this register to be used for general purpose tasks by compiling pointers relative to the current frame. In order to allow a native sampling profiler to collect stack traces of JIT-compiled Java stack frames, the JVM provides the option *PreserveFramePointer*, that instructs the HotSpot compiler to preserve the frame pointer in the RSP register [21]. This might incur a small overhead that needs to be accounted for in the evaluation of the overhead of the profiler.

For both the LLSS and MLSS approaches, the overhead and accuracy are expected to be dependent on the sampling rate, and for higher sampling rates, *perf* is shown to perform better than *async-profiler*, however, the performance of *perf* is dependent on the operating system and runtime environment. The expected overhead for both of these profilers is approximately 5% for a sample rate in the order of 1 kHz [18]. However, for big data applications 1 kHz might prove to be too high. On a distributed system with many multi-core machines, the number of samples collected can rapidly exceed practical boundaries at kHz sample rates.

3.3 Evaluation

In order to make a decision on which profiling approach is the most suitable, we want to know the overhead that is introduced by the code instrumentation of these approaches. To evaluate this, the instrumentation required for these approaches was implemented in Spark, and the SFJA query was run on a TPC-H dataset with scale factor 10. The results of this are shown in fig. 3.2. The LLSS approach does not require instrumentation, and therefore was not included in this evaluation.

The TTI approach, which uses the system clock to measure time spent in operators, might be feasible for coarser grained applications, but in this case the clock resolution and its overhead are an order of magnitude greater than the expected execution time of a small operator. As such, we do not consider this to be a feasible approach. The IAS approach proved to be complex to implement, and still introduces a relatively large overhead into the pipeline. The method splitting instrumentation of the MLSS has an overhead similar to the instrumentation-aided approach, however the implementation is straightforward using existing tools. Based on these findings, the decision was made not to pursue the TTI and the IAS approaches further, and to further evaluate the MLSS and LLSS approaches.

Finally, to compare the MLSS approach to the LLSS approach, both of these approaches were prototyped against the SFJA query, and the profiling results were compared. Furthermore, a profile of uninstrumented code was collected with using *async-profiler*, which is representative of the current state of the art. In order to perform this comparison, the profilers were run on the SFJA query executed against same TPC-H dataset, stored as a Parquet file, with scaling factor of 10. The interpretation of the results can be seen in fig. 3.3. From these results, the breakdown of performance cost per operator in the SFJA query is as shown in table 3.1.

From the results of the uninstrumented code profile, we can draw some small conclusions about the execution costs of the query. What can be observed is that the *scan* operator takes

	Uninstrumented	MLSS	LLSS
Scan	45.1%	58.3% (30.7% + 27.6%)	31.30%
Filter	54.9%	13.2%	51.22%
Join		8.13%	7.52%
Agg.		5.08%	9.15%

Table 3.1: A breakdown of the costs per operator, as collected by each of the three profilers. In parentheses, the unattributed costs for the MLSS query are shown.

up a large portion of time, and that the majority of the work is done on code that is expected to be invoked by the *filter*, *join* or *aggregate* operators. However, it is difficult to obtain an exact per-operator breakdown of the work without expert knowledge of the generated code. A small portion of the code is attributed to the method *doAggregateWithKeys*, without any further detail. This is labeled as *unidentified overhead* in the figure. Furthermore, some time is spent in the *dematerialization*, that is, loading values from the input columns into registers, which is not easily attributed to one operator. Within code generation, however, this is invoked by the operator which first loads this attribute into a register, and ideally should be attributed to that operator.

As expected, the MLSS approach shows a more detailed breakdown of the operators than the sampling of uninstrumented code. A separate stack frame is shown for each of the operators, and the runtime of these operators is calculated as the number of samples in these frames. However, a large portion of the collected samples is attributed to the method `scan_doProduce_0`, without any further breakdown of the costs. This *unidentified overhead* fraction makes up 27.6% of the overall costs, making it significantly greater than that of the uninstrumented code. The body of the loop that this overhead is attributed to performs the small tasks of deserialization, and invoking the child operator. Since this is expected to be much less work than the actual execution of the invoked partial pipeline, we do not believe that the attribution of this much overhead to *scan* is credible. Furthermore, this breakdown shows the problem caused by eager dematerialization required for this instrumentation approach, more than doubling the cost of this dematerialization, which it then also adds to the observed cost of the *scan* operator.

For the LLSS approach, we observe that this approach can provide highly detailed profiles of the generated query plans, and can attribute most of the work done to a specific operator in the pipeline. Only small fractions of the code in distinct operators remain unexplained, and can be realistically attributed to execution of inline code. Furthermore, it attributes the dematerialization performed to invoke the filter operator to the *consume*-section of that operator, whereas the MLSS approach evaluates dematerialization eagerly, attributing all of this work to the bottom-most operator that implements a *produce* section, which in this case is the *scan* operator.

We conclude from these observations that the MLSS and LLSS approaches provide feasible methods of obtaining profiles for generated code pipelines in Spark SQL queries. MLSS, however, shows a lack of detail in the results, its attribution of a large number of samples to the *scan* operator lacks credibility, and furthermore, it requires functional modification of the generated code in order to collect these stack traces. The LLSS approach suffers significantly less from these problems, and therefore we believe these results to be more reliable. In chapter 5 we will investigate the differences between these approaches further.

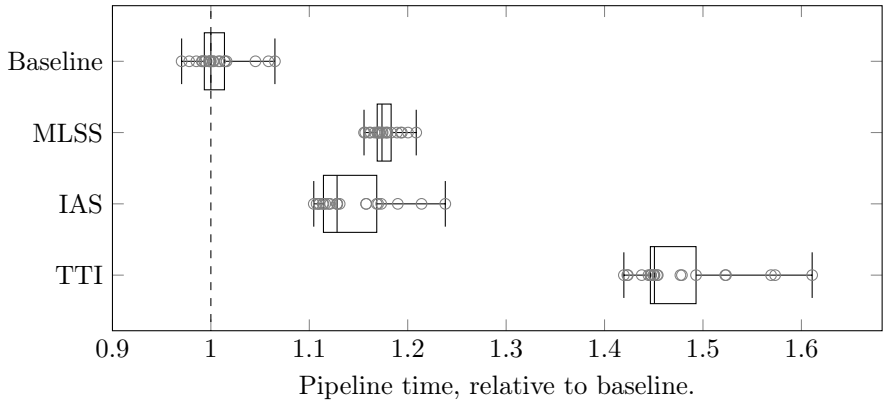
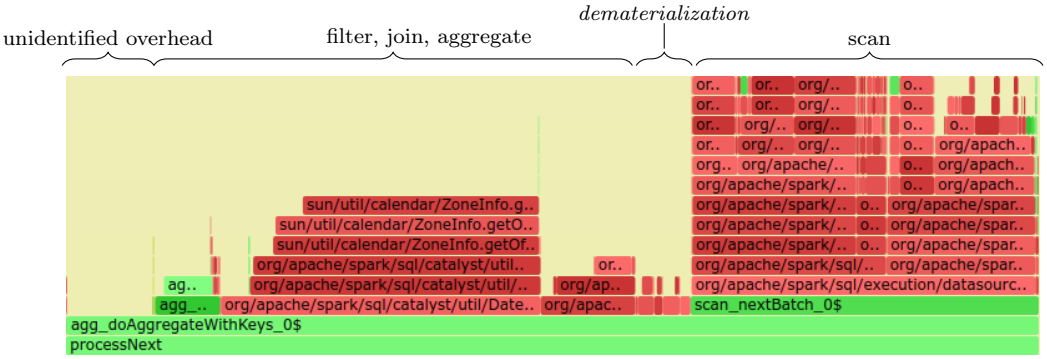
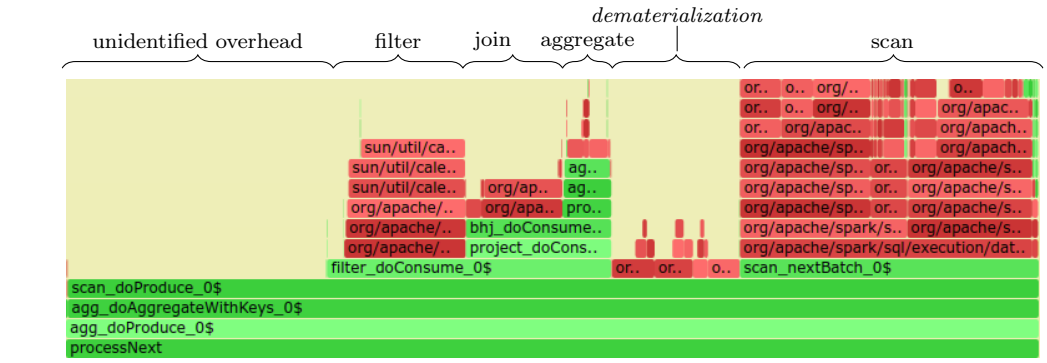


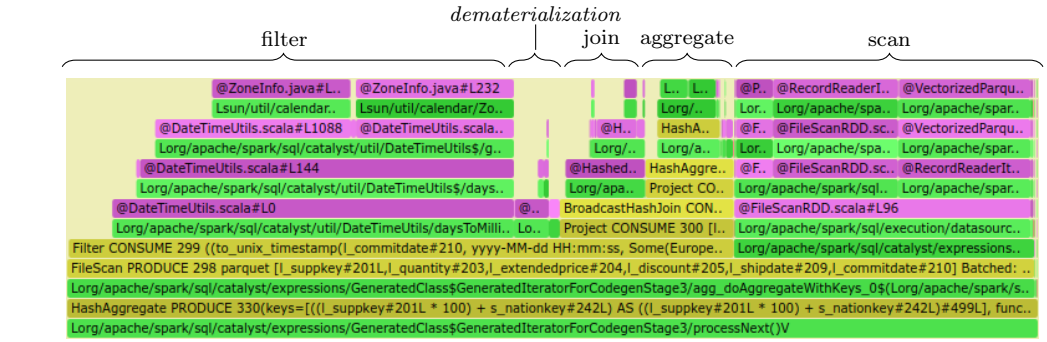
Figure 3.2: The effect of the different instrumentation approaches on CPU time. This benchmark was run on a single executor, against the canonical SFJA query, on a TPCH dataset with scale factor 10.



Method sampling with uninstrumented code, *async-profiler*



MLSS using method-level sampling with operators split into methods, *async-profiler*



LLSS using *perf_ event*, JVMTI agent

Figure 3.3: Three fragments of CPU profiles of the same SFJA pipeline, contrasting simple profiles of unmodified Spark, code splitting using *async-profiler*, and the *perf_ event* approach.

Chapter 4

LLSS system design

To investigate the accuracy and performance of the line-level stack sampling approach, the LLSS profiler was implemented in Spark. For the purposes of evaluation, the open source branch for Spark version 2.4 is used. In this chapter, we will discuss the implementation choices that were made.

4.1 Overview

The profiler operates as a stand-alone service on each executor, and the executor is responsible for starting and stopping the profiler. Profiling data is collected locally, and is only collected on the driver side on request by the user.

The system consists of two elements: the JVMTI, and the profiler. Figure 4.1 shows how these two elements interact with the executor. The profiler collects call stacks of the native execution of JIT-compiled methods in the JVM. This profiler is implemented using a profiler manager which runs in the JVM, which, for each thread that requests a profiling session to start, opens a new *perf_ event* file descriptor, and forks the JVM process to start a child process that is responsible for the profiling of that thread, passing it the file descriptor.

The agent is implemented using the JVMTI interface provided by the JVM, allowing it to intercept events within the JVM that let us collect metadata of the JIT-compiled methods that is necessary to identify the context of an instruction pointer.

4.2 Agent

To collect metadata on method compilation, the JVMTI is used [20]. This API allows for the collection of the events *ClassPrepare*, *CompiledMethodLoad*, and *CompiledMethodUnload*. The JVM fires these events every time before a new class is loaded, a method is compiled, or a compiled method is unloaded, respectively, which can be captured by registering an agent that provides event handlers for these methods.

The JVMTI-agent is the only writer for metadata. The data is written to a shared map consisting of the loaded classes and compiled methods. To avoid locking in the JVM, which would potentially slow down the execution, these shared maps are constructed as skip-lists in a manner that is mostly lock-free, such that searching an element in the map is a safe operation for the profiler, but reading the element is not atomic, and might result in a corrupted stack in the sample. However, since the chances of this occurring are extremely small, and the results are

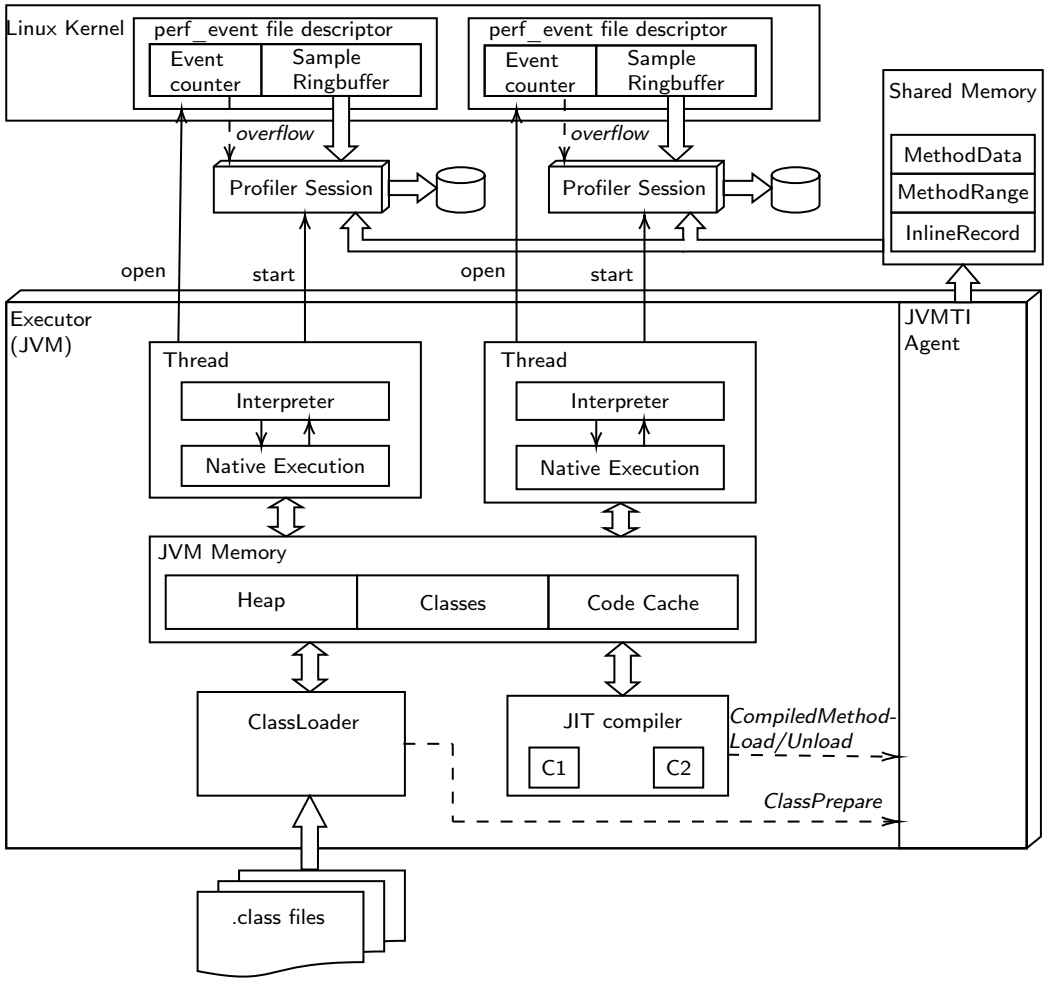


Figure 4.1: Overview of the LLSS profiler design.

that one sample out of many might be corrupted, this is an acceptable risk.

Table 4.1 shows the data collected for the methods of classes on *ClassPrepare*, the compiled method address ranges for each *CompiledMethodLoad* event, and the inlined methods for each instruction in the compiled method. These events are used to build an ordered set of `InlineRecord`s, which are defined as a stack of inlined method frames. These records are then stored as a map, mapping instruction pointer, ordered by instruction pointer. The collected metadata is stored in a region of virtual memory that is shared between the JVMTI agent and the profiler, allowing the agent to produce metadata and the profiler to access the produced metadata at runtime.

4.3 Profiler

In order to obtain profiles at a granularity fine enough to be able to aggregate these at an operator level, existing profiling tools for the JVM are not sufficient. Therefore, we seek to extend the work of existing profiling approaches, in order to allow profiling at a granularity small enough to identify the overhead at the level of lines of code.

When a new Java thread requests a profiler to start, an independent profiler session is started by the manager for that thread by forking from the JVM process, that samples, reads, and writes results independently from all other threads. This has the advantage that results can be written asynchronously, and no concurrency issues arise from interaction between these profilers. Communication with the manager is performed through a UNIX-socket that acts as a message queue, which allows for the initialization of the session, and for starting and stopping a profiling run for the Java thread.

Stack sampling As a means of stack sampling, when starting the new session, the profiler manager opens a new *perf_ event* file descriptor, using the Linux system call *perf_ event_ open* [14] which samples performance events only for the thread requesting the session. The manager can be passed a sample rate in Hz, which is passed to the *perf_ event* file handle. The configuration options passed to the system call are the following:

type	PERF_TYPE_SOFTWARE
config	PERF_COUNT_SW_TASK_CLOCK
sample_type	PERF_SAMPLE_CALLCHAIN
sample_freq	*from configuration*
freq	1

This sets the handle up to listen to software-generated task clock events, at a configured sampling frequency, and providing call chains as its sample results. After the perf_event file descriptor is opened, a new `ProfilerSession` is opened, which is passed the file descriptor. The ProfilerSession then opens the required memory mapped buffer that allows it to read the sampled call chains.

Event handling In order to notify the profiler of newly collected samples, the *perf_ event* API allows a signal handler to be attached to a UNIX signal that is fired each time the counter hits its overflow value. This signal can be used to obtain the sampled stack from the allocated by the *perf_ event_ open* action.

The JVM, however, uses UNIX signals internally to perform various tasks, and for most signals, signal handlers are installed for the process that is to be profiled. Replacing these handlers would interfere with the operation of the JVM. To prevent this, the choice was made to

Class methods

Type	Field	Description
uint64	Method ID	A unique identifier that remains the same as long as the containing class remains loaded.
char[]	Signature	Provides a unique string that describes the class path, method, and signature of the method.
pair<int,int>[]	Line number table	A relation mapping bytecode indices to line numbers of the source code, collected during class compilation.

Compiled method range

Type	Field	Description
uint64	Method ID	The Method ID of the class method.
void *	Start address	The start address of the compiled code, which together with the end address define the address range of the compiled method's machine code.
void *	End address	The end address of the compiled code.

Inlined methods

Type	Field	Description
void *	Instruction Pointer	The instruction pointer of the generated code.
uint64[]	Method IDs	For each frame in the inline stack, provides a method ID for the method that was inlined.
int32[]	Bytecode Indices	For each frame in the inline stack, provides a bytecode index describing the bytecode in the method that generated the address.

Table 4.1: Data collected by the JVMTI agent of the profiler

listen to overflow signals from a separate, isolated process, and to collect samples asynchronously. This has the added advantage that the execution of the JVM is not halted for the duration of the event handler, and its state remains unaffected by the sampling events. Furthermore, much like *async-profiler*, thread-specific buffers allow for the asynchronous collection of samples without concurrency issues, and without the need for locking.

When a profiling session starts, the profiler starts a new thread that listens to events. When it receives a signal, the listening thread performs the following actions:

1. From the memory mapped *perf* file, the stack chain is read, and the tail is advanced
2. For each instruction pointer in the call chain, look up the method ID
3. For each method ID that is found, look up the inline stack, method signatures, and line numbers for the instruction pointer
4. Inject the stack, signature, and line numbers in the call chain
5. In a local hash map, find or create a new counter for the obtained call chain, and increment it.

Integration As a proof of concept, the profiler was integrated into Apache Spark by adding start- and end- event triggers into the task executor, which the profiler subscribes a listener to. Figure 4.2 shows how the different parts of the profiler interact to start and stop the collection of task profiles. This approach allows for easy integration of different profilers or other monitoring tools, but this does require some changes to the executor's task execution loop. It is desirable to be able to keep the results of tasks from different stages apart, since these might be part of different queries. However, at the same time, the output should be limited to prevent overhead caused by heavy I/O use. In order to do so, the profiler is passed metadata of the task, which includes the stage ID. The profiler then only writes its results each time a task start event occurs, and the stage ID of the previous task is different from the next task.

The results are written to a temporary directory, along with the class files that were generated, which are made available for later reference. The format that is used is the same as that used as the *collected* output format of *async-profiler*, consisting of a plain text document containing lines of semicolon-separated methods in the call chain. Inlined methods are treated the same as standalone methods. In addition to methods, however, if line number information is collected for the method, the method is followed by a string `<filename>.java@L<line no>`.

The semicolon-separated list of methods is followed by a space, and finally by the number of collected samples for that call chain as decimal number. This can later be used in post-processing, to produce results aggregated on the operator level.

Code generation modifications The generated code modifications for this approach are minimal, and do not affect executed statements in the Java code. Instead, the code is annotated with block comments, delineating the generated code regions, describing whether the region is part of a *consume* or *produce* operation, and a unique ID of the operator that generated the block. Spark does not keep a unique ID for operators, however this is trivially added by adding an automatically incrementing ID for each construction of an operator instance. To illustrate this approach of declaring regions using comments, listing 4.1 shows the *consume* operation of the broadcast operator seen in listing 2.1, annotated with region comments. Note that unlike code splitting, this approach can attribute the dematerialization of the input values is entirely to the correct operator.

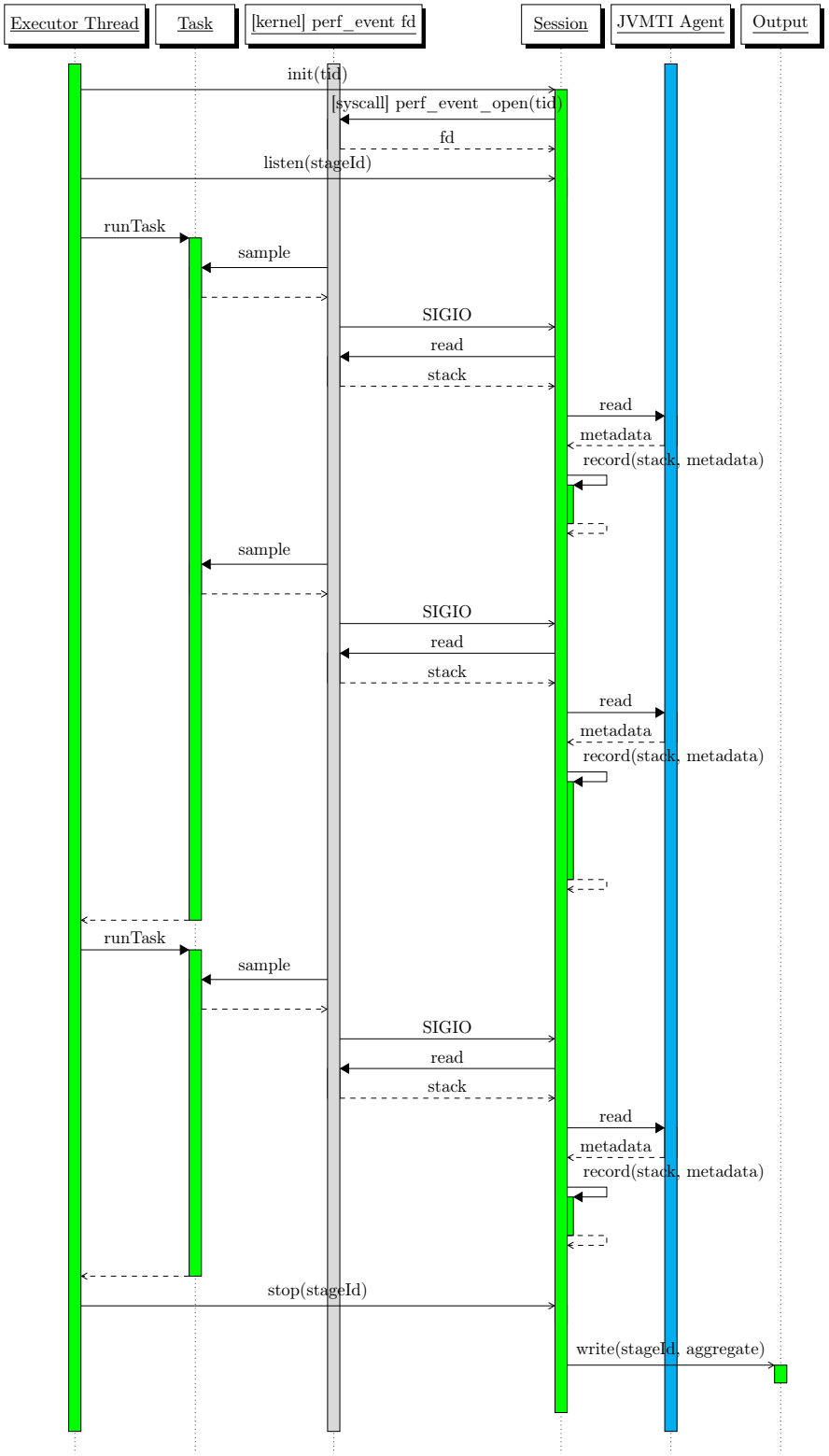


Figure 4.2: Sequence diagram of the collection of a profile for a sequence of tasks of the same stage.

Listing 4.1: A comment-delimited operator.

```
1 /* BeginCodeRegion(CONSUME_7) */
2 // <child operator produces a row>
3 int join_key = row.getInt(0);
4 Row broadcast_row = broadcastRelation.getRowForKey(join_key)
5 if (broadcast_row != null) {
6     // <parent operator consumes both rows>
7 }
8 /* EndCodeRegion(CONSUME_7) */
```

This information can be parsed trivially using regular expressions to collect a stack of operators for each line of the source code. To be able to access this source code, two more modifications have to be made. Firstly, by default within Spark, the Janino compiler compiles the code without debugging metadata included. This debugging metadata is what the JVMTI uses to obtain information about the source file, as well as information on the lines of code that generated the bytecode. Thus, the compiler must be configured to output this information as class metadata such that the JVMTI can access it. Secondly, the generated code is kept as a string in the memory of the executors, and is discarded after compilation. In order to access this source code, we write the information to a temporary file, with a unique name that is passed to the class metadata. This allows the profiler to access the source code after query execution.

This modification is not strictly necessary. The Java class file format allows for arbitrary metadata to be specified, which would allow a mapping from bytecode to operators to be written directly into the class file, which could then be intercepted using the `ClassFileLoadHook` event in the JVMTI, without modifying the generated code. However, since the Spark code generator does not collect information on the origin of lines of code, this would require a large overhaul of the code generation. Because of the limited payoff and large impact of such an overhaul, the decision was made to delimit operator regions with comment blocks instead. Another advantage of this approach is that arbitrary code regions can be specified for each operator, for example a scan operator can define a region for reading a batch of code.

Runtime statistics During the execution of the SQL query, events are produced by the driver and the executors throughout the execution of the jobs that belong to this query. These events include data on which stages are part of which SQL query, and which tasks belong to these stages, as well as the execution plan of the query that was performed, including a mapping of operator ID to operators. Furthermore, various statistics are collected for each task, including the total CPU time spent working on the task. The data is collected for each SQL query, and relayed to the driver, through a SQL event listener that implements the *SparkListener* interface [1]. The collected data is stored for later use.

Aggregation The results produced by the profiler are partially aggregated on method ID and code location. Further aggregation is performed offline, at the request of the user. To obtain the profile of a specific SQL query execution, each executor is instructed to perform the following operations:

Collection The executors collect profiles by stage ID. The collected runtime statistics are used to identify the execution stages that were part of the the SQL query.

[1] https://spark.apache.org/docs/2.3.1/api/java/org/apache/spark/scheduler/SparkListener.htm

Annotation After the stage's profiles are collected, the temporarily stored generated code classes are used to obtain the code region annotations generated by the modified code generator. For each operator that matches to a line number of a frame in the collected stacks, the matching region is inserted into the call stack, and recorded as a frame, such that the final stack consists of a semicolon separated list of methods, code locations, and operator data.

Aggregation All the profiles of the collected stages are collected from the executors by the driver, and summed, aggregating on unique stacks.

Operator resolution By matching the operator ID recorded in the code region names, the operator data is retrieved from the SQL event listener, and inserted into the profile data.

Simplification As a final, optional step, the aggregated profile can be simplified by taking all non-operator frames out of the stack, and re-aggregating to obtain a profile consisting of just the operators in the pipeline. This may be helpful for non-expert users who are not acquainted with the Java code.

The final output follows the input requirements for the *flamegraph.pl* tool [8], consisting of a semicolon separated list of elements representing the call stack, followed by a number of samples. The output is formatted such that the bottom-most element in the stack is the stage ID, and followed by a list of methods, file locations, and operators. If the final simplification step is applied, only the operators are included in the stack.

The bottom chart in fig. 3.3 was constructed using this approach, and using *flamegraph.pl* to render the final graph.

Chapter 5

Evaluation

In section 1.2, a number of criteria were identified that a profiler should fulfill. This chapter discusses the detailed evaluation of the LLSS profiler, and compares the accuracy to that of the MLSS profiler.

5.1 Experimental setup

The accuracy experiments are performed on a single executor on a Linux machine. For accuracy measurement, this is assumed to be representative of distributed execution, because the profiling of the tasks is performed on a per-task basis, and within Spark's architecture, individual tasks are run on a single machine regardless of the cluster setup. Since a local machine lends itself well for these experiments, the following setup is used:

CPU	Intel Haswell i7 4790k
#Executors	1
Operating System	Linux
Kernel Version	5.0.0
OS Distribution	Ubuntu 18.04
Spark version	Apache Spark v2.4.5 (04b3e0e)

For the overhead experiments, a single machine setup is not representative. Due to the handling of I/O in the system, granularity and scheduling play a large role in the performance. Furthermore, in order to perform these experiments, large datasets are required. In order to run these experiments, the system was ported to the Databricks Runtime (DBR) 5.5, which is a fork of Apache Spark 2.4, and a cloud cluster was setup with the following properties:

Cloud provider	Amazon AWS EC2
Instance type	i3-xlarge
#Executors	10
Operating System	Linux
Kernel Version	4.4.0
OS Distribution	Ubuntu 16.04
Spark version	DBR v5.5 (Apache Spark v2.4.3)

5.2 Accuracy

In this section, we will discuss the evaluation of the accuracy of the LLSS and MLSS approaches.

5.2.1 Methodology

Performance profiling is subject to uncertainty that is inherent to the execution of a program in the JVM. During the execution of a pipeline, the generated Java code may be subject to multiple phases of optimization and compilation. Within the generated code pipeline, this entails that optimization and reordering may take place across operator boundaries, and multiple operations and different control flow structures may be fused together. This makes attribution of the CPU-time spent in these regions challenging. Additionally, the CPU might perform out-of-order execution, and the instruction pointer that is reported by a sampling profiler might not be representative of the different instructions that are in the re-order buffer of the CPU.

Furthermore, since a Java program is JIT-compiled, and the JVM performs runtime profiling and introspection to estimate the effect of optimizations on the running code. By interfering with the running program, the JVM may be inhibited from building an accurate profile, which might influence the performance of operators in the pipeline, and introduce inaccuracies into the profile. Moreover, if the code is instrumented, this might also interfere with the JVMs ability to optimize methods, by preventing reordering and in some cases even JIT-compilation of the instrumented methods.

In order to provide a definition of accuracy that allows for the evaluation of a profiler for the canonical query, we will use the canonical SFJA query plan, and introduce artificial overhead, such that specific operators in the SFJA pipeline are stressed. By providing different parameterizations of this query plan, such that differences in performance between these parameterizations can be isolated to a single operator, it might be possible to create an oracle that provides a notion of an accurate profile.

Care must be taken to choose queries and parameterizations such that only one operator in the pipeline is affected. In particular, parameterizations that change size or layout of the data introduce an unpredictable load on all operators, due to extra costs introduced in scans, as well as in materialization of values. To achieve this, various strategies might be explored, such as leveraging short circuiting in string matching in order to vary the required work without changing string size, introducing user defined functions into the query, and changing the size of the broadcast table in the join operation.

First, a baseline is established for the evaluation of the accuracy, by running the unmodified canonical query. The setup used was the TPC-H dataset, with a scaling factor of 10, on which the canonical query was run, using a single executor with 8 cores. Since the execution of pipelines is independent per executor, it is reasonable to assume this is representative for distributed execution. To establish the baseline, the query was run 5 times without modification, but with the LLSS profiler enabled to ensure a uniform and fair comparison, and the mean baseline running time $\overline{T}$ is taken. Furthermore, these profiles were sampled, and for each operator, the average number of samples $\overline{s}_{op}$ was collected, to obtain a baseline profile.

Then, to evaluate the accuracy, a number of queries are run with overhead injected at one single operator the SFJA query, for all operators, except for the *scan*, since this operator only takes the table as input and cannot easily be altered. This way four different experiments are run with following added overhead:

Reference	No overhead was introduced.
Altered filter	Overhead was injected in the WHERE clause of the query, on the input expression for the filter predicate.

Altered join Overhead was injected in the join key expression, at the streaming side of the broadcast hash join. The expression should include columns from both tables, to prevent the evaluation from being pushed down into the query plan subtrees of the separate tables.

Altered agg. Overhead was injected in the expression of the GROUP BY clause. Like the join operator, the expression includes columns from both tables.

The way this overhead is introduced, is by wrapping these expressions in an identity function that causes some significant computation to be performed, but which has no impact on the eventual value of the expression.

This gives us three different queries with an artificial overhead introduced in the chosen operator *op*, which is profiled to obtain a distribution s across the 4 operators, as well as time spent outside the operators.

$$S = s_{scan} + s_{filter} + s_{join} + s_{aggregate} + s_{other}$$

Furthermore, the total CPU time T_{op} for the query is known to spark, which lets us obtain the real time difference caused by the inserted overhead in *op*:

$$t_{op} = T_{op} - \overline{T}$$

This time can then be compared to the time observed by sampling the pipeline. To calculate this observed time, the total CPU time T spent inside the pipeline is known to spark, allowing o_{op}, an estimate of the time in an operator, to be calculated:

$$o_{op} = T_{op} \frac{s_{op} - \overline{s}_{op}}{S}$$

To obtain a metric for the accuracy of the profile for the operator, the relative error E is calculated:

$$E = \frac{o_{op} - t_{op}}{t_{op}}$$

For each of the operators, the above experiment is run 5 times to obtain a sample of relative errors. Of this, the mean absolute relative error can be taken to obtain a metric of accuracy for the profiler. All samples recorded outside of the stage belonging to the SFJA pipeline are discarded, and only the samples within the relevant stage were analysed. The CPU time is calculated as the time spent within the SFJA pipeline stage.

5.2.2 Results

To gain an understanding of the accuracy of the profiler at different frequencies, the experiment described above was run for the configurations of 2 Hz, 20 Hz, and 200 Hz. At higher frequencies, the collected profiles become too large to handle effectively, so these frequencies were not investigated.

An overview of each individual run and the distribution of collected samples over the operators can be seen in fig. 5.1. As the sampling frequency increases, the repeatability of the sampling method increases greatly. Furthermore, the overhead that could not be attributed to one of the four operators is extremely low. Detailed results for each of the runs can be found in appendix A. By taking the mean of $|E|$ of these benchmarks, a metric for the accuracy of the profiler is obtained, which is shown in fig. 5.3.

At 2 Hz, the total number of samples collected for the reference query is around an average of 57, making for a distribution of samples that sometimes reaches 0 for some operators. This makes for unreliable results, as is can be seen in the collected profiles, which sometimes point to the incorrect bottleneck operator. At higher frequencies, the results become more reliable, and at 200 Hz, the result is accurate within a 10% margin of error, even for the smaller aggregation operator, and the correct bottleneck operator is consistently found.

To compare these results to the MLSS approach, the same experiments were run using the MLSS profiler, the results of which can be seen in figures 5.2 and 5.3, as well as in appendix A. At a sampling rate of 200 Hz, the MLSS approach is shown to have an overall higher mean absolute relative error for all three modified operators. In particular, the results for the altered join stand out. Where the LLSS profiler slightly overestimates the introduced overhead in the join operator, the MLSS consistently underestimates this overhead by 8%-13%, depending on the sampling rate.

Furthermore, the MLSS approach attributes a much higher cost to the FileScan operator, which might at least partly be explained by both the extraneous dematerialization of unused fields, as well as the misattribution of samples that occur within the FileScan's *produce* section, as was shown in fig. 3.3. In the *reference* configuration, this results in the MLSS approach estimating the average cost of the *scan* operator to be 52.4% of the total pipeline time, as opposed to 21.1% as measured by the LLSS approach. This corresponds to our findings in chapter 3. This additional cost appears to be constant regardless of introduced overhead, and therefore our accuracy metric is unaffected by this behavior. We do, however, consider this to be a source of inaccuracy of the profiler.

We also observe that in the *altered filter* configuration, the MLSS profiler often fails to collect any samples at all, or only collects a handful of samples, of the join operator. Comparing this to the same workload in the *reference* run, where the average number of samples collected at 200 Hz is 223, we see that the MLSS profiler is in disagreement with itself depending on the exact structure of the code that it is profiling.

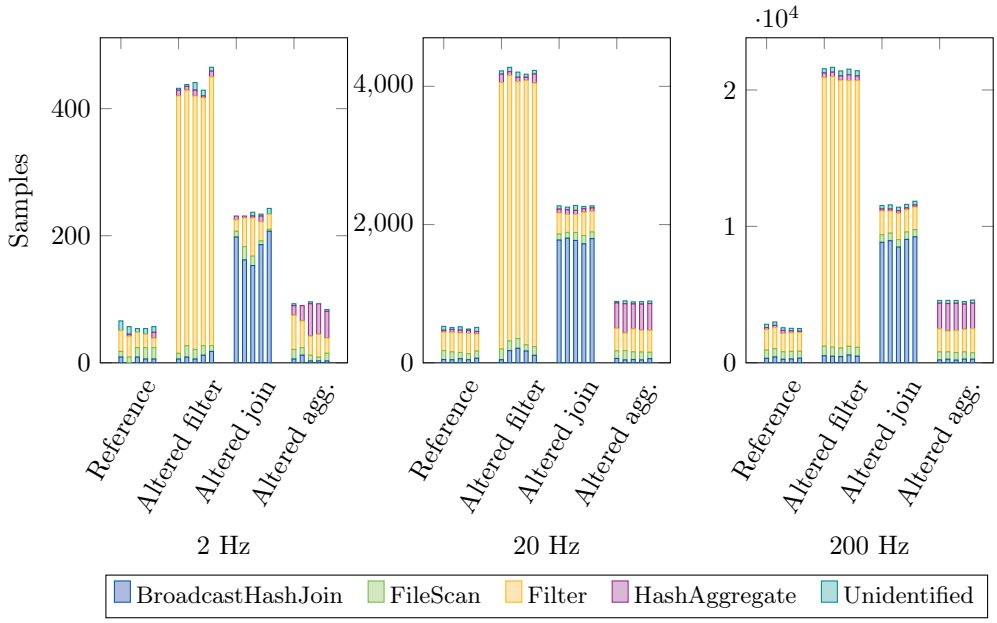


Figure 5.1: Distribution of samples in individual runs of the SFJA query using the LLSS profiler, for the reference run and the 3 different operators, for the three different sampling frequencies.

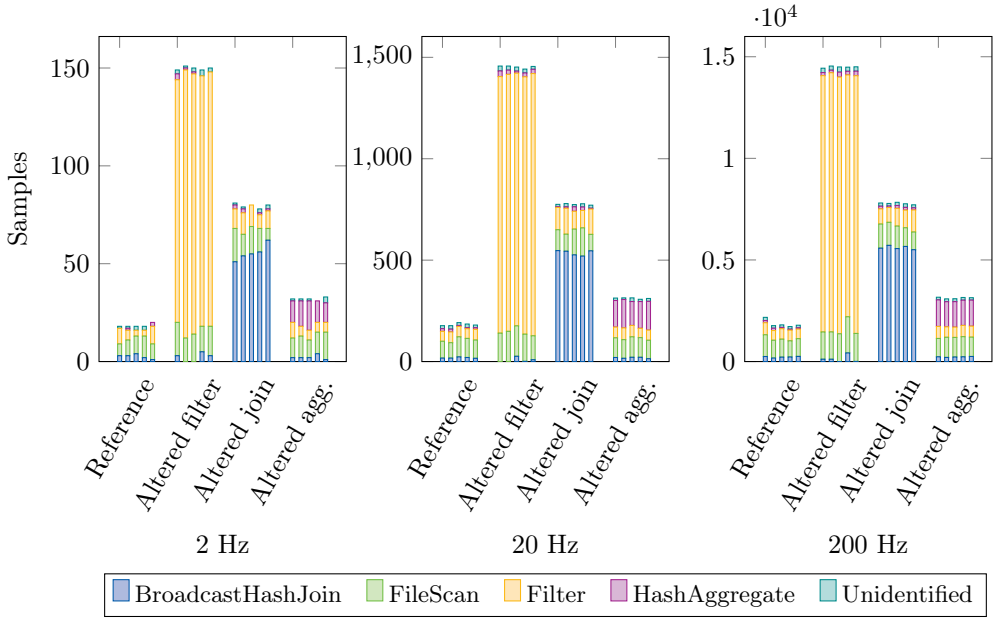


Figure 5.2: Distribution of samples for individual runs of the SFJA query using the MLSS profiler. Notice the high cost attributed to FileScan compared to fig. 5.1.

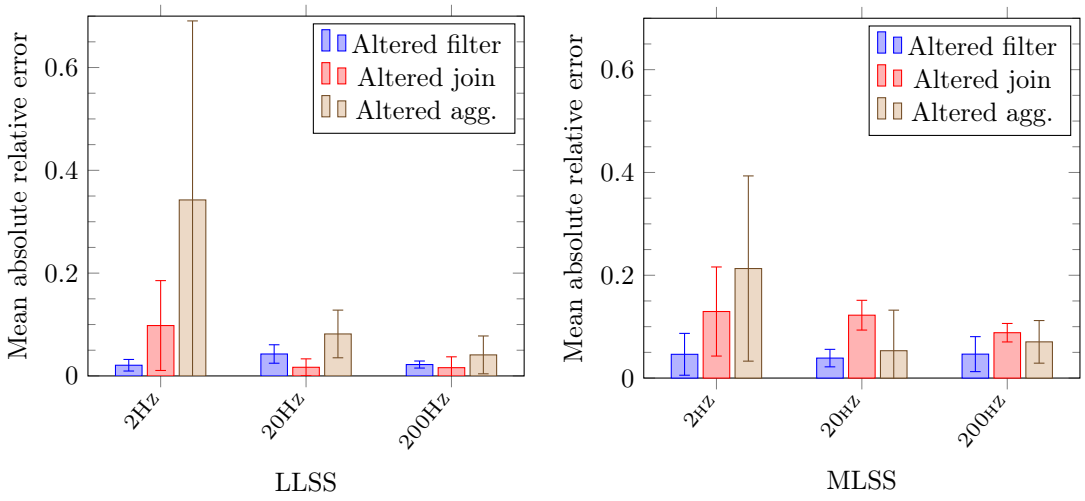


Figure 5.3: Mean absolute relative error for the different configurations and bottleneck operators for the LLSS profiler. The error bars show a 95% confidence interval of the mean.

5.3 Overhead

Besides accuracy, we are also interested the overhead of the LLSS approach. In this section, we discuss the evaluation of this overhead.

5.3.1 Methodology

Within generated code, register allocation becomes a challenge as the number of columns increases. Since some approaches affect register allocation behavior of the JVM, a benchmark representative of queries on large numbers of columns is desired. Therefore, a benchmark with a suitable number of columns should be used to evaluate the impact of this approach. The TPC-H benchmark [27] provides a set of queries that are suitable for this. As a representative cross section of this set of queries, the following queries are used [12]:

Q1	Aggregation with 8 aggregation functions
Q3	Join
Q6	Global aggregation with selective filter
Q9	Large join
Q18	High-cardinality aggregation

For the overhead analysis, the wall clock time of query execution is measured, from the time the query execution is started by the driver, to the time the driver receives the result.

5.3.2 Results

For each of the configurations of 2 Hz, 20 Hz and 200 Hz, on a TPC-H dataset with scaling factor 100, a benchmark was run using the six queries. 10 iterations of this benchmark were run. For

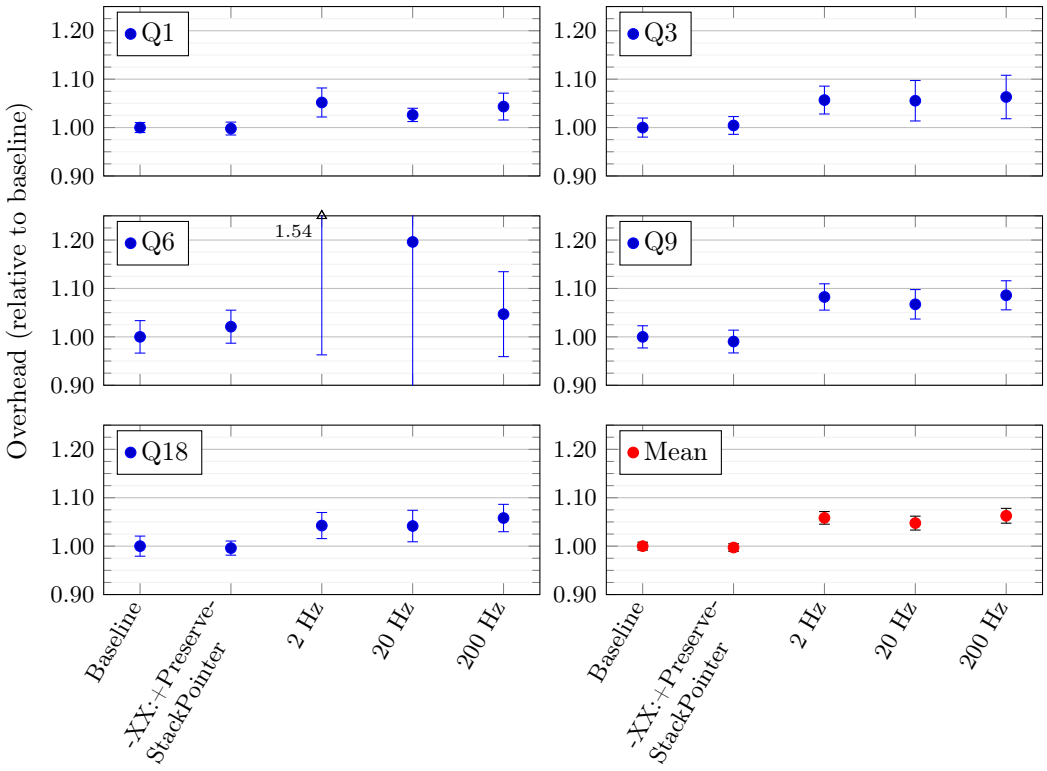


Figure 5.4: Overhead of the LLSS profiler at different sampling frequencies, for the 5 selected TPC-H queries, with 95%-confidence error bars.

the 200 Hz configuration, one outlier measuring 263.2 seconds for Q18 was removed from the results, and 9 out of 10 samples were used in the evaluation. The results can be found in fig. 5.4.

The results for Q1, Q3, Q8 and Q18 show an overhead of less than 10%, tending towards 5%. Most of this overhead is constant, and unaffected by sampling frequency. In the performance results for this set of queries, a leap is seen between the baseline and the LLSS profiler with the 2 Hz configuration, and an increase of the sampling rate does not cause this overhead to increase. Furthermore, the results for Q6, which consists of much more fine-grained tasks, and shows a much greater overhead. This is indicative of a significant overhead that is not caused by the sampling itself, but by the interaction between the calling thread and the profiler. These interactions, which were discussed in chapter 4, include method collection via the JVMTI, inter-process communication via message passing, and I/O via shared memory maps. Although measures were taken to mitigate the performance impact these interactions might introduce, better results might be achieved by further optimization of the profiler. The *PreserveStackPointer* flag is shown to have negligible effect on execution times.

5.4 Threats to validity

By introducing an artificial workload into the operators of the query pipelines, the code of the pipeline method is altered, which will result in unintended changes in the execution of the pipeline. Among other things, these are caused by register allocation, caching, optimizations, and JIT-compilation in the JVM. This might affect operators other than the target operator, and introduce some unobservable error in the oracle, and introduces an error in the accuracy evaluation.

As mentioned in section 5.3, one outlier was removed from the set of benchmark results for the overhead. Although the cause of this outlier has not been found, it has been observed that very rarely, running the LLSS profiler causes Spark executors to freeze for a significant amount of time, not accepting new tasks until the inactive threads are terminated by Spark. This might point to a race condition in the task handling of the profiler. This may have affected the results of the performance evaluation, however since this effect has a large impact, this would certainly result in an obvious outlier, such as the one that was removed.

Another potential threat to the validity of the performance results stems from a known performance regression in the JVMTI, affecting Oracle JDK 8 and OpenJDK 8 [1]. This regression leads to poor performance of the collection of metadata of loaded classes, which the LLSS profiler uses to map native stacks to JVM stacks. By preloading these classes using a warm-up run, this effect is largely mitigated. Generated classes, however, cannot be preloaded, and will be loaded during the execution of the benchmark. Since this only concerns a handful of classes per execution, this is unlikely to have a significant effect on the results.

[1] https://bugs.openjdk.java.net/browse/JDK-8185348

Chapter 6

Conclusion

In this thesis we discussed the different performance aspects that are relevant for a profiler for fine-grained SQL query pipelines in Apache Spark, and we developed a profiler for obtaining SQL profiles of query executions run in Apache Spark, which provides accurate results for compiled query pipelines. In this chapter, we will discuss these findings, and the answers to the research questions we have found.

The main contribution of this thesis, is that this is first work to achieve a feasible and accurate approach of profiling compiled query pipelines in Apache Spark. Using our LLSS approach, we can overcome the challenges posed by the multiple stages of optimization and compilation of the Spark SQL execution model, and we can obtain profiles of queries that provide a breakdown of the performance cost per operator. Using this approach, we can correctly and accurately point out the problematic operators in a query plan that has been executed by Spark.

RQ1. What approaches for obtaining performance profiles can be used for profiling the execution of generated code of an SFJA query?

We have identified various approaches in previous research that focus on collecting profiles in Java applications, and investigated the finest granularity that these profilers can achieve. Furthermore, we have introduced a novel approach of aggregating samples of native stack traces on the level of operators in generated code pipelines.

Approaches To answer the first subquestion of RQ1, we have explored four approaches of profiling query pipelines in Apache Spark. The first two approaches, which we named TTI and IAS, are based on instrumentation of the generated code. That is, instructions collecting data are introduced in regions of interest in the generated code, to collect timing information and obtain a profile. TTI uses the Java method `System.nanoTime` to obtain timing information, and the IAS approach keeps a variable in memory to keep the currently executing operator in memory. The latter approach was not pursued further due to implementation difficulties with synchronization, added uncertainty due to caching, and no clear benefits.

Two other approaches are based on stack sampling. The first, MLSS, uses the industry tool *async-profiler* to obtain stack traces of Java code in the JVM at the granularity of Java methods. In order to be able to aggregate these results at an operator level, the generated code pipeline is split into separate methods, which incurs an overhead.

For the final approach, LLSS, we extended on the approach used in the implementation of *async-profiler* to build a new profiler. This profiler uses native stack sampling using

the Linux kernel API *perf_ events*, to obtain highly detailed profiles. These are then aggregated at the level of operators, by collecting metadata on each compilation step required to compile the query plan from the abstract tree, via Spark's code generation, Java compilation, and finally JVM JIT compilation, to native instructions. This approach does not require modification of functional generated code, and only injects annotations as block comments to denote regions of interest in the code for each generated operator.

Granularity The granularity of the approaches we have identified in previous literature was shown to be too broad for profiling unmodified SFJA pipelines as these are currently implemented in Spark. Many profilers are capable of profiling on the level of Java methods, but do not offer the option to profile at a finer level. For the MLSS approach, we have shown that profiles of the SFJA pipeline can be aggregated at an operator level by splitting the pipeline into smaller method is a feasible way of obtaining these profiles, at the cost of an increased overhead. The results of this approach, however, are not very detailed, and suffer from an observer effect caused by the modified source code. Furthermore, within the SFJA pipeline, there is an apparent over-representation of samples in the scan's *produce* section.

For the LLSS approach, we have shown that *perf-event*, in combination with detailed information on JVM JIT compilation supplied by the JVMTI, provide a method of obtaining fine-grained profiles at the level of lines of Java code, or even at the level of native machine instructions. Using this approach, we built a profiler that produces profiles at a fine enough granularity to be able to map these back to operators in the SFJA pipeline.

Furthermore, we have shown that the granularity of the TTI approach, due to the dependency on the system clock provided by the operating system, has a lower bound of approximately 30 ns on Linux. This is too large to obtain reliable profiles at the level of SQL operators that might take less than a nanosecond to execute.

RQ2. How can the accuracy of a profiler be defined within the scope of the SFJA query pipelines?

To obtain the accuracy of the profiler for the SFJA pipeline, we have created an oracle based on an SFJA query. By introducing an artificial overhead into the distinct operators in the generated pipeline, a ground truth can be established on the time spent executing this artificial overhead in a certain operator in the pipeline. The profile collected by a profiler can then be tested against this ground truth, to obtain the accuracy of the profiler.

Using this approach, we have evaluated the accuracy of the MLSS and LLSS profiling approaches. We have found that the LLSS approach attributes the introduced overhead to the correct operator with a mean absolute error below 10%, and tending to 5% at a 200Hz sampling rate. For the MLSS approach, the mean absolute error is slightly higher.

Furthermore, the MLSS approach consistently overestimates the *scan* operator in a generated code pipeline. We believe that this is due to the attribution of unidentified runtime costs in the pipeline, as well as the costs of eager dematerialization of input tuples, to this operator.

Based on these results, we believe that the LLSS approach is more accurate and more reliable than the MLSS approach.

RQ3. What is the impact of the integration of the identified approaches in Apache Spark?

In order to understand the impact of integration of the profiler into Apache Spark, we have looked at two aspects, namely implementation, and overhead.

The code changes to the Spark codebase itself are minimal. Only a few event handlers need to be introduced to be able to communicate to the profiler when execution of a task starts, and when it ends.

To evaluate the overhead of the LLSS approach has over query execution, we have run a subset of the queries in the TPC-H benchmark with and without the LLSS profiler. We have observed that the LLSS profiler introduces a constant overhead for the tasks that are part of the query execution, which is mostly independent of sampling frequency. These are possibly caused by the interprocess communication and I/O of the profiler. This amounts to approximately 5%-10% of the execution cost for most queries, and with the exception of queries consisting of very small tasks, such as Q6 in the TPC-H benchmark, this overhead remains within 10% of the baseline cost.

In conclusion, we have shown that of the explored approaches the LLSS approach outperforms the other approaches in terms of accuracy. At a sampling rate of 200Hz, the results are within 10% accurate. Although our implementation suffers from a relatively high overhead even at low sample rates, we believe this overhead to be constant regardless of sampling rate, and to be at least partly caused by the implementation choice of running the profiler in separate processes. Through further optimization of I/O, and perhaps refactoring the profiler to not run from within a separate process, we believe that this can be improved drastically, and the general approach that we have introduced provides a blueprint for a profiler that allows for the collection of highly accurate profiles, against a low cost, for compiled SQL queries in Apache Spark.

6.1 Future work

LLSS and MLSS overhead In this thesis, we have observed that the LLSS approach has a constant overhead of approximately 5% for most TPC-H queries. However, we have not investigated a way to break down this performance impact, and to find the root cause of this overhead, even at the low sampling frequency of 2 Hz. A closer evaluation of this overhead might provide insight into the performance issues that we have encountered.

Due to time constraints, we have focused most of the evaluation in this thesis on the LLSS approach, which showed the most promising results. We have observed that for the SFJA query, a 17.6% overhead is introduced by splitting operators into methods. Based on this result we have decided not to pursue this method any further, and we did not collect information on the overhead of the MLSS approach. Further investigation of performance impact of this approach would provide more insight into the feasibility of this approach.

Task sampling As an extension to the discussed approaches, the spark environment could be modified to allow for the possibility of sampling on the separate tasks in the job that is to be profiled. One approach of sampling is sampling on regular tasks, running only a subset of the tasks with the profiler enabled, and the rest without profiling or instrumentation. This is expected to limit the overhead of the profiling, while also limiting the effect this has on the accuracy of the collected data.

There are some pitfalls associated with this approach, that have to be taken into account. Many queries are subject to skew or other biases in the separate tasks, which, if sampling not done carefully, could result in bias in the resulting profile. Furthermore, care has to be taken to avoid sampling the tasks in such a way that slots running non-profiled tasks are left idle while waiting for the profiled tasks, which are likely to take longer than non-profiled tasks.

In order to avoid the latter issue, it might be interesting to explore the possibility of sampling tasks by exploiting speculative task scheduling in Spark to duplicate selected tasks, and

disconnecting them from the scheduling DAG. By doing so, it can be guaranteed that there are no other tasks that depend on them, and profiling of these tasks might eliminate any additional latency, as long as sufficient resources are available to perform these additional tasks in parallel.

Continuous monitoring The goal of this thesis was to investigate profiling approaches that might give users insight into the performance of their SQL queries, by obtaining fine-grained profiles of generated code pipelines. We have not investigated how such a profiling method can be used to continuously profile user's queries, and provide insight on past query executions. This provides an opportunity for further research which would be highly relevant to industry use cases, where there is a high demand for tools which provide insight into past issues through continuous monitoring.

Bibliography

[1] AMD. Processor programming reference (ppr) for amd family 17h model 01h, revision b1 processors. `http://developer.amd.com/wordpress/media/2017/11/54945_PPR_Family_17h_Models_00h-0Fh.pdf`, 2017.

[2] Apple. *DTRACE(1), BSD General Commands Manual*, 18.7.0 edition, 2019.

[3] Michael Armbrust, Ali Ghodsi, Matei Zaharia, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, and Michael J. Franklin. Spark SQL. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, volume 142, pages 1383–1394, New York, New York, USA, dec 2015. ACM Press.

[4] Peter A Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Proceedings of the 2005 CIDR Conference*, volume 5, pages 225–237, 2005.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[6] YourKit GmbH. YourKit Java Profiler^{TM}. `https://www.yourkit.com/features/`.

[7] G. Graefe and W.J. McKenna. The Volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*, pages 209–218. IEEE Comput. Soc. Press, 1993.

[8] Brendan Gregg. The Flame Graph. *ACM Queue*, 14(2):91–110, 2016.

[9] Brendan Gregg. The pmcs of ec2: Measuring ipc. `https://web.archive.org/web/20200108105904/http://www.brendangregg.com/blog/2017-05-04/the-pmcs-of-ec2.html`, 2017.

[10] Intel. Intel VTune Profiler. `https://software.intel.com/en-us/vtune`, 2019.

[11] Stephen Kell, Danilo Ansaloni, Walter Binder, and Lukáš Marek. The JVM is not observable enough (and what to do about it). In *Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages - VMIL '12*, page 33, New York, New York, USA, 2012. ACM Press.

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

[13] John Levon. Oprofile. https://oprofile.sourceforge.io/, 2004.

[14] Linux Foundation. *perf_ event_ open(2), Linux Programmer's Manual*, 5.0.0 edition, 2019.

[15] Wes McKinney and contributors. Pandas: Python Data Analysis Library. `https://pandas.pydata.org/`.

[16] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of Java profilers. *ACM SIGPLAN Notices*, 45(6):187, 2010.

[17] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. In *Proceedings of the VLDB Endowment*, volume 4, pages 539–550, 2011.

[18] Andy Nisbet, Nuno Miguel Nobre, Graham Riley, and Mikel Luján. Profiling and Tracing Support for Java Applications. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 119–126, 2019.

[19] Oracle. Java Platform Standard Edition 8 Documentation. `https://docs.oracle.com/javase/8/docs/`.

[20] Oracle. JVM Tool Interface. `https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html`, 2007.

[21] Oracle and OpenJDK contributors. OpenJDK 8 HotSpot, advancedThresholdPolicy.hpp. `http://hg.openjdk.java.net/jdk8u/jdk8u/hotspot/file/2b2511bd3cc8/src/share/vm/runtime/advancedThresholdPolicy.hpp`.

[22] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot tm server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium*, volume 1, 2001.

[23] Andrei Pangin et al. *async-profiler*. `https://github.com/jvm-profiling-tools/async-profiler`.

[24] R Development Core Team. R: A language and environment for statistical computing. `https://www.r-project.org/`.

[25] Johannes Rudolph, Nitsan Wakart, et al. *perf-map-agent*. `https://github.com/jvm-profiling-tools/perf-map-agent`.

[26] Aibek Sarimbekov, Andreas Sewe, Walter Binder, Philippe Moret, and Mira Mezini. JP2: Call-site aware calling context profiling for the Java Virtual Machine. *Science of Computer Programming*, 79:146–157, 2014.

[27] Transaction Processing Performance Council. *TPC Benchmark^{TM} H*, 2.18.0 edition, 2018.

[28] Igor Veresov. Tiered compilation in hotspot jvm. `https://www.slideshare.net/maddocig/tiered`, 2013.

[29] Richard Warburton et al. *honest-profiler*. `https://github.com/jvm-profiling-tools/honest-profiler`.

[30] John Whaley. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande - JAVA '00*, pages 78–87, New York, New York, USA, 2000. ACM Press.

[31] Thomas Willhalm and Roman Dementiev. Intel® performance counter monitor - a better way to measure cpu utilization. `https://software.intel.com/en-us/articles/intel-performance-counter-monitor`, 2012.

[32] Stephen Yang, Seo Jin Park, and John Ousterhout. Nanolog: A nanosecond scale logging system. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 335–350, Boston, MA, July 2018. USENIX Association.

[33] Matei Zaharia, Michael J Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, Ion Stoica, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, and Shivaram Venkataraman. Apache Spark. *Communications of the ACM*, 59(11):56–65, oct 2016.

[34] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. *8th USENIX Symposium on Operating Systems Design and Implementation*, 8(4):29–42, 2008.

[35] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A real-time side-channel attack detection system in clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 118–140. Springer, 2016.

[36] Marcin Zukowski. *Balancing vectorized query execution with bandwidth-optimized storage.* PhD thesis, University of Amsterdam, 2009.

Appendix A

Accuracy benchmark results

Run	Test operator	Operator Name					Sum of samples	CPU Time (s)	*perf* estimate (s)	Expected overhead (s)	Error
		Join	Scan	Filter	Aggregate	Unidentified					
		s_{join}	s_{scan}	s_{filter}	s_{aggr}		$s_{total} = \Sigma s$	T_{op}	o_{op}	t_{op}	E
1	Reference	393	633	1410	264	336	3036	11.642			
2	Reference	360	540	1239	213	204	2556	8.843			
3	Reference	306	453	1380	204	213	2556	8.790			
4	Reference	243	543	1341	162	228	2517	8.695			
5	Reference	303	507	1431	84	198	2523	8.768			
	Reference mean	328.2	565.8	1424.4	153.6	214.8		9.695			
6	Altered Filter	426	591	**19242**	465	309	21033	70.577	61.215	62.606	-2.22%
7	Altered Filter	549	567	**19050**	387	315	20868	70.017	61.620	62.868	-1.98%
8	Altered Filter	591	651	**18996**	465	345	21048	70.552	60.988	62.007	-1.64%
9	Altered Filter	861	618	**18798**	276	354	20907	70.214	60.531	62.484	-3.12%
10	Altered Filter	624	603	**19101**	537	321	21186	70.924	60.800	62.109	-2.11%
11	Altered Join	**8667**	606	1602	279	195	11349	38.210	28.608	29.124	-1.77%
12	Altered Join	**8943**	522	1257	312	195	11229	37.843	29.006	29.254	-0.85%
13	Altered Join	**8793**	501	1548	294	210	11346	38.309	27.547	28.840	-4.48%
14	Altered Join	**8724**	564	1476	282	246	11292	37.979	29.387	29.466	-0.27%
15	Altered Join	**8700**	540	1560	252	237	11289	38.113	29.957	30.156	-0.66%
16	Altered Agg.	264	558	1605	**1857**	204	4488	15.345	5.877	5.924	-0.79%
17	Altered Agg.	327	591	1503	**1809**	216	4446	15.178	6.307	5.859	+7.65%
18	Altered Agg.	237	549	1599	**1866**	198	4449	15.188	6.297	5.900	+6.74%
19	Altered Agg.	255	522	1536	**2013**	189	4515	15.408	5.758	5.584	+3.12%
20	Altered Agg.	279	552	1470	**1872**	201	4374	14.950	5.822	5.951	-2.17%

Table A.1: Single executor SFJA benchmark with sampling frequency 200 Hz for the LLSS profiler.

Run	Experiment	Operator Name					Sum of samples	CPU Time (s)	*perf* estimate (s)	Expected overhead (s)	Error
		Join	Scan	Filter	Aggregate	Unidentified					
		s_{join}	s_{scan}	s_{filter}	s_{aggr}		$s_{total} = \Sigma s$	T_{op}	o_{op}	t_{op}	E
21	Reference	48	129	267	27	57	528	9.12			
22	Reference	45	114	282	42	27	510	8.90			
23	Reference	60	90	282	42	48	522	8.90			
24	Reference	45	87	291	48	18	489	8.65			
25	Reference	66	105	246	36	60	513	8.88			
	Reference mean	52.8	105	273.6	39	42	512.4	8.889			
26	Altered Filter	45	156	**3852**	123	48	4224	71.07	60.20	62.18	-3.17%
27	Altered Filter	177	141	**3837**	57	63	4275	71.86	59.90	62.97	-4.88%
28	Altered Filter	210	141	**3717**	63	75	4206	70.43	57.66	61.54	-6.31%
29	Altered Filter	171	90	**3825**	45	45	4176	70.29	59.77	61.40	-2.64%
30	Altered Filter	108	126	**3813**	132	51	4230	70.92	59.34	62.03	-4.34%
31	Altered Join	**1776**	87	306	54	48	2271	38.28	29.04	29.39	-1.17%
32	Altered Join	**1803**	84	258	69	39	2253	37.74	29.31	28.85	+1.62%
33	Altered Join	**1770**	114	264	57	72	2277	38.44	28.98	29.54	-1.90%
34	Altered Join	**1722**	120	336	51	33	2262	38.01	28.04	29.11	-3.68%
35	Altered Join	**1797**	96	300	48	30	2271	38.26	29.38	29.37	+0.05%
36	Altered Agg.	63	111	324	369	21	**888**	15.25	5.66	6.36	-10.92%
37	Altered Agg.	42	135	255	423	42	**897**	15.22	6.51	6.33	+2.92%
38	Altered Agg.	48	111	333	360	30	**882**	15.22	5.54	6.33	-12.51%
39	Altered Agg.	42	114	315	378	36	**885**	15.14	5.79	6.25	-7.21%
40	Altered Agg.	60	93	315	390	36	**894**	15.42	6.05	6.53	-7.27%

Table A.2: Single executor SFJA benchmark with sampling frequency 20 Hz for the LLSS profiler.

Run	Test operator	Operator Name					Sum of samples	CPU Time (s)	*perf* estimate (s)	Expected overhead (s)	Error
		Join	Scan	Filter	Aggregate	Unidentified					
		s_{join}	s_{scan}	s_{filter}	s_{aggr}		$s_{total} = \Sigma s$	T_{op}	o_{op}	t_{op}	E
41	Reference	9	9	33		15	66	10.20	n/a	n/a	n/a
42	Reference		9	33	3	12	57	9.57	n/a	n/a	n/a
43	Reference	9	15	24		6	54	8.78	n/a	n/a	n/a
44	Reference	6	18	21		9	54	8.92	n/a	n/a	n/a
45	Reference	6	18	15	9	9	57	8.96	n/a	n/a	n/a
	Reference mean	7.5	13.8	25.2	6	10.2	57.6	9.284	n/a	n/a	n/a
46	Altered Filter	6	9	**405**	9	3	432	72.99	64.170	63.70	+0.73%
47	Altered Filter	9	18	**402**	6	3	438	73.84	63.520	64.55	-1.60%
48	Altered Filter	6	15	**399**	9	12	441	73.30	62.127	64.01	-2.94%
49	Altered Filter	12	15	**390**	3	9	429	72.28	61.460	62.99	-2.43%
50	Altered Filter	18	9	**423**	9	6	465	76.82	65.717	67.53	-2.69%
51	Altered Join	**198**	9	18	6		231	38.79	31.986	29.50	+8.42%
52	Altered Join	**162**	21	45	3		231	39.74	26.577	30.45	-12.73%
53	Altered Join	**153**	15	60	3	6	237	39.62	24.323	30.33	-19.82%
54	Altered Join	**186**	6	30	9	3	234	40.20	30.664	30.91	-0.81%
55	Altered Join	**207**	3	24		9	243	39.60	32.514	30.31	+7.24%
56	Altered Agg.	6	15	**54**	15	3	93	15.66	1.515	6.37	-76.22%
57	Altered Agg.	12	12	**42**	24		90	15.41	3.081	6.12	-49.67%
58	Altered Agg.	3	9	30	**51**	3	96	15.18	7.117	5.90	+20.66%
59	Altered Agg.	3	6	36	**48**		93	15.45	6.976	6.16	+13.19%
60	Altered Agg.	3	12	24	**42**	3	84	15.08	6.465	5.80	+11.46%

Table A.3: Single executor SFJA benchmark with sampling frequency 2 Hz for the LLSS profiler. The profiler failed to identify the correct bottleneck operators at runs 56 and 57.

Run	Test operator	Operator Name					Sum of samples	CPU Time (s)	*perf* estimate (s)	Expected overhead (s)	Error
		Join	Scan	Filter	Aggregate	Unidentified					
		s_{join}	s_{scan}	s_{filter}	s_{aggr}		$s_{total} = \Sigma s$	T_{op}	o_{op}	t_{op}	E
1	Reference	248	1070	592	107	158	2175	9.736			
2	Reference	172	876	479	114	127	1768	9.663			
3	Reference	217	885	514	63	126	1805	9.329			
4	Reference	225	796	527	46	129	1723	9.170			
5	Reference	253	873	473	63	126	1788	9.528			
	Reference mean	223	900	517	78.6	133.2	1851.8	9.485			
6	Altered filter	115	1340	**12614**	152	215	14436	71.917	60.265	62.432	-3.47%
7	Altered filter	114	1344	**12756**	124	204	14542	71.872	60.490	62.387	-3.04%
8	Altered filter	1	1365	**12634**	243	254	14497	72.023	60.198	62.537	-3.74%
9	Altered filter	424	1780	**11901**	179	203	14487	72.212	56.745	62.727	-9.54%
10	Altered filter	6	1377	**12683**	224	215	14505	72.441	60.760	62.956	-3.49%
11	Altered join	**5580**	1186	748	131	158	7803	39.652	27.222	30.167	-9.76%
12	Altered join	**5717**	1134	729	71	129	7780	39.428	27.843	29.943	-7.01%
13	Altered join	**5560**	1107	872	139	158	7836	38.973	26.544	29.488	-9.98%
14	Altered join	**5664**	920	869	134	177	7764	39.120	27.415	29.635	-7.49%
15	Altered join	**5504**	870	1086	106	148	7714	39.455	27.011	29.970	-9.87%
16	Altered agg.	234	902	592	**1307**	130	3165	15.904	6.173	6.419	-3.84%
17	Altered agg.	203	991	514	**1238**	139	3085	16.265	6.113	6.779	-9.84%
18	Altered agg.	223	964	509	**1253**	143	3092	15.878	6.031	6.393	-5.66%
19	Altered agg.	234	991	553	**1243**	132	3153	16.256	6.003	6.770	-11.33%
20	Altered agg.	250	951	542	**1281**	119	3143	15.830	6.056	6.345	-4.55%

Table A.4: Single executor SFJA benchmark with sampling frequency 200 Hz for the MLSS profiler.

Run	Test operator	Operator Name					Sum of samples	CPU Time (s)	*perf* estimate (s)	Expected overhead (s)	Error
		Join	Scan	Filter	Aggregate	Unidentified					
		s_{join}	s_{scan}	s_{filter}	s_{aggr}		$s_{total} = \Sigma s$	T_{op}	o_{op}	t_{op}	E
21	Reference	17	83	49	13	15	177	9.080			
22	Reference	17	76	53	14	17	177	9.137			
23	Reference	23	99	51	5	14	192	9.942			
24	Reference	20	94	48	5	18	185	9.498			
25	Reference	16	90	53	8	13	180	9.134			
	Reference mean	18.6	88.4	50.8	9	15.4	182.2	9.358			
26	Altered filter		140	**1265**	28	24	1457	73.294	61.080	63.936	-4.47%
27	Altered filter		149	**1267**	21	20	1457	73.511	61.362	64.153	-4.35%
28	Altered filter	26	150	**1246**	10	20	1452	72.973	60.067	63.614	-5.58%
29	Altered filter	1	134	**1269**	19	19	1442	72.153	60.955	62.795	-2.93%
30	Altered filter	9	118	**1293**	21	14	1455	73.162	62.461	63.803	-2.10%
31	Altered join	**547**	103	110	3	12	775	39.154	26.696	29.796	-10.41%
32	Altered join	**544**	84	128	7	16	779	39.174	26.421	29.816	-11.39%
33	Altered join	**526**	127	88	22	11	774	39.059	25.606	29.701	-13.79%
34	Altered join	**520**	139	86	17	16	778	39.429	25.411	30.071	-15.50%
35	Altered join	**546**	81	125	4	15	771	39.119	26.759	29.760	-10.09%
36	Altered agg.	20	97	53	**131**	12	313	15.877	6.189	6.519	-5.07%
37	Altered agg.	16	92	58	**140**	8	314	16.039	6.691	6.680	+0.16%
38	Altered agg.	21	101	56	**118**	18	314	15.974	5.545	6.615	-16.18%
39	Altered agg.	21	97	46	**132**	11	307	15.785	6.324	6.427	-1.59%
40	Altered agg.	14	91	50	**142**	14	311	15.937	6.815	6.578	+3.60%

Table A.5: Single executor SFJA benchmark with sampling frequency 20 Hz for the MLSS profiler.

Run	Test operator	Operator Name					Sum of samples	CPU Time (s)	*perf* estimate (s)	Expected overhead (s)	Error
		Join	Scan	Filter	Aggregate	Unidentified					
		s_{join}	s_{scan}	s_{filter}	s_{aggr}		$s_{total} = \Sigma s$	T_{op}	o_{op}	t_{op}	E
41	Reference		3	8	6	1	18	9.831			
42	Reference	1	3	5	8	1	18	9.651			
43	Reference		4	3	9	2	18	9.340			
44	Reference		2	3	11	2	18	9.295			
45	Reference	2	1	9	8		20	9.080			
	Reference mean	1.5	2.6	5.6	8.4	1.5	18.4	9.439			
46	Altered filter	3	3	**124**	17	2	149	74.787	59.428	65.347	-9.06%
47	Altered filter	1		**137**	12	1	151	74.747	65.045	65.308	-0.40%
48	Altered filter	1		**133**	14	2	150	74.371	63.166	64.932	-2.72%
49	Altered filter		5	**128**	13	3	149	74.570	61.258	65.131	-5.95%
50	Altered filter		3	**130**	15	2	150	74.206	61.542	64.767	-4.98%
51	Altered join	**2**	51	10	17	1	81	40.484	24.191	31.045	-22.08%
52	Altered join	**2**	54	11	11	1	79	40.041	26.052	30.602	-14.87%
53	Altered join		55	11	14		80	39.996	26.198	30.557	-14.27%
54	Altered join	**1**	56	7	12	2	78	40.310	27.597	30.871	-10.61%
55	Altered join	**1**	62	9	6	2	80	40.137	29.802	30.698	-2.92%
56	Altered agg.	11	2	8	**10**	1	32	16.007	4.752	6.567	-27.64%
57	Altered agg.	13	2	5	**11**	1	32	16.247	5.839	6.808	-14.23%
58	Altered agg.	15	2	5	**9**	1	32	16.389	6.914	6.950	-0.51%
59	Altered agg.	11	4	5	**11**		31	16.030	4.912	6.590	-25.46%
60	Altered agg.	10	1	5	**14**	3	33	16.280	4.193	6.840	-38.70%

Table A.6: Single executor SFJA benchmark with sampling frequency 2 Hz for the MLSS profiler.

Appendix B

Profile-annotated code pipeline

Listing B.1: Plan subtree for the annotated code

```
[101] WholeStageCodegen
 |- [97] HashAggregate(keys=[], functions=[partial_sum(l_extendedprice#204)])
     |- [96] Project [l_extendedprice#204]
         |- [95] Filter (isnotnull(l_shipdate#209) && (l_shipdate#209 > 9496))
             |- [94] FileScan parquet [l_extendedprice#204,l_shipdate#209] lineitem
```

Listing B.2: A profile of a generated code stage of a simple query

```
Time %  1   public Object generate(Object[] references) {
 0.00%  2     return new GeneratedIteratorForCodegenStage1(references);
 0.00%  3   }
 0.00%  4
 0.00%  5   /*wsc_codegenStageId*/
 0.00%  6   final class GeneratedIteratorForCodegenStage1 extends BufferedRowIterator {
 0.00%  7     private Object[] references;
 0.00%  8     private scala.collection.Iterator[] inputs;
 0.00%  9     public java.nio.ByteBuffer stateBuffer;
 0.00% 10     private long totalTime_97_0;
 0.00% 11     private boolean agg_initAgg_0;
 0.00% 12     private boolean agg_bufIsNull_0;
 0.00% 13     private double agg_bufValue_0;
 0.00% 14     private long agg_totalTime_96_0;
 0.00% 15     private long project_totalTime_95_0;
 0.00% 16     private long filter_totalTime_94_0;
 0.00% 17     private long scan_scanTime_0;
 0.00% 18     private int scan_batchIdx_0;
 0.00% 19     private boolean agg_agg_isNull_2_0;
 0.00% 20     private boolean agg_agg_isNull_4_0;
 0.00% 21     private OnHeapColumnVector[] scan_mutableStateArray_2 = new
                  ↪ OnHeapColumnVector[2];
 0.00% 22     private UnsafeRowWriter[] scan_mutableStateArray_3 = new UnsafeRowWriter
                  ↪ [5];
 0.00% 23     private ColumnarBatch[] scan_mutableStateArray_1 = new ColumnarBatch[1];
 0.00% 24     private scala.collection.Iterator[] scan_mutableStateArray_0 = new scala.
                  ↪ collection.Iterator[1];
 0.00% 25
 0.00% 26     public GeneratedIteratorForCodegenStage1(Object[] references) {
 0.00% 27       this.references = references;
 0.00% 28     }
 0.00% 29
 0.00% 30     public void init(int index, scala.collection.Iterator[] inputs) {
 0.00% 31       partitionIndex = index;
 0.00% 32       stateBuffer = java.nio.ByteBuffer.allocateDirect(8);
 0.00% 33       this.inputs = inputs;
 0.00% 34       totalTime_97_0 = 0L;
 0.00% 35
 0.00% 36       agg_totalTime_96_0 = 0L;
 0.00% 37       project_totalTime_95_0 = 0L;
 0.00% 38       filter_totalTime_94_0 = 0L;
 0.00% 39       scan_mutableStateArray_0[0] = inputs[0];
 0.00% 40       scan_mutableStateArray_3[0] = new UnsafeRowWriter(2, 0);
 0.00% 41       scan_mutableStateArray_3[1] = new UnsafeRowWriter(2, 0);
 0.00% 42       scan_mutableStateArray_3[2] = new UnsafeRowWriter(1, 0);
 0.00% 43       scan_mutableStateArray_3[3] = new UnsafeRowWriter(1, 0);
 0.00% 44       scan_mutableStateArray_3[4] = new UnsafeRowWriter(1, 0);
 0.00% 45
 0.00% 46     }
 0.00% 47
 0.00% 48     private void agg_doAggregateWithoutKey_0() throws java.io.IOException {
 0.00% 49       // initialize aggregation buffer
 0.00% 50       agg_bufIsNull_0 = true;
 0.00% 51       agg_bufValue_0 = -1.0;
 0.00% 52
 0.00% 53       /* CodeRegion(PRODUCE_96) */
 0.00% 54       /* Project */
 0.00% 55       /* CodeRegion(PRODUCE_95) */
 0.00% 56       /* Filter */
 0.00% 57       /* CodeRegion(PRODUCE_94) */
 0.00% 58       /* Scan parquet */
 0.00% 59       if (scan_mutableStateArray_1[0] == null) {
 0.00% 60         scan_nextBatch_0();
 0.00% 61       }
 0.00% 62       while (scan_mutableStateArray_1[0] != null) {
 0.00% 63         int scan_numRows_0 = scan_mutableStateArray_1[0].numRows();
 0.00% 64         int scan_localEnd_0 = scan_numRows_0 - scan_batchIdx_0;
 0.00% 65         for (int scan_localIdx_0 = 0; scan_localIdx_0 < scan_localEnd_0;
                      ↪ scan_localIdx_0++) {
```

```
 1.18%   66         int scan_rowIdx_0 = scan_batchIdx_0 + scan_localIdx_0;
 0.00%   67         /* CodeRegion(CONSUME_95) */
 0.00%   68         /* Filter */
 0.00%   69         do {
 0.00%   70           boolean scan_isNull_1 = scan_mutableStateArray_2[1].isNullAt(
                                ↪ scan_rowIdx_0);
11.53%   71           int scan_value_1 = scan_isNull_1 ? -1 : (scan_mutableStateArray_2
                                ↪ [1].getInt(scan_rowIdx_0));
 0.00%   72
 0.00%   73           if (!(!scan_isNull_1)) continue;
 0.00%   74
 0.09%   75           boolean filter_value_2 = false;
 0.00%   76           filter_value_2 = scan_value_1 > 9496;
 2.22%   77           if (!filter_value_2) continue;
 0.00%   78
 2.13%   79           ((SQLMetric) references[2] /* numOutputRows */).add(1);
 0.00%   80
 0.00%   81           /* CodeRegion(CONSUME_96) */
 0.00%   82           /* Project */
 0.00%   83           /* CodeRegion(CONSUME_97) */
 0.00%   84           /* HashAggregate */
 0.00%   85           boolean scan_isNull_0 = scan_mutableStateArray_2[0].isNullAt(
                                ↪ scan_rowIdx_0);
 1.56%   86           double scan_value_0 = scan_isNull_0 ? -1.0 : (
                                ↪ scan_mutableStateArray_2[0].getDouble(
                                ↪ scan_rowIdx_0));
 0.00%   87
 0.00%   88           agg_doConsume_0(scan_value_0, scan_isNull_0);
 0.00%   89
 0.00%   90           /* /CodeRegion(CONSUME_97) */
 0.00%   91           /* /CodeRegion(CONSUME_96) */
 0.84%   92         } while(false);
 0.00%   93
 0.00%   94         /* /CodeRegion(CONSUME_95) */
 0.00%   95         // shouldStop check is eliminated
 0.00%   96         }
 0.51%   97         scan_batchIdx_0 = scan_numRows_0;
 0.00%   98         scan_mutableStateArray_1[0] = null;
 0.29%   99         scan_nextBatch_0();
 0.00%  100       }
 0.17%  101       ((SQLMetric) references[1] /* scanTime */).add(scan_scanTime_0 / (1000 *
                                ↪  1000));
 0.00%  102       scan_scanTime_0 = 0;
 0.00%  103
 0.00%  104       /* /CodeRegion(PRODUCE_94) */
 0.00%  105       /* /CodeRegion(PRODUCE_95) */
 0.00%  106       /* /CodeRegion(PRODUCE_96) */
 0.00%  107     }
 0.00%  108
 0.02%  109     private void scan_nextBatch_0() throws java.io.IOException {
 0.00%  110       long getBatchStart = System.nanoTime();
48.48%  111       if (scan_mutableStateArray_0[0].hasNext()) {
 0.00%  112         scan_mutableStateArray_1[0] = (ColumnarBatch)scan_mutableStateArray_0
                                ↪ [0].next();
 0.69%  113         ((SQLMetric) references[0] /* numOutputRows */).add(
                                ↪ scan_mutableStateArray_1[0].numRows());
 0.00%  114         scan_batchIdx_0 = 0;
 0.01%  115         scan_mutableStateArray_2[0] = (OnHeapColumnVector)
                                ↪ scan_mutableStateArray_1[0].column(0);
 0.00%  116         scan_mutableStateArray_2[1] = (OnHeapColumnVector)
                                ↪ scan_mutableStateArray_1[0].column(1);
 0.00%  117
 0.00%  118       }
 0.02%  119       scan_scanTime_0 += System.nanoTime() - getBatchStart;
 0.00%  120     }
 0.00%  121
 0.00%  122     /* CodeRegion(CONSUME_97) */
 0.00%  123     /* HashAggregate */
 0.12%  124     private void agg_doConsume_0(double agg_expr_0_0, boolean
                                ↪ agg_exprIsNull_0_0) throws java.io.IOException
                                ↪ {
 0.00%  125       // do aggregate
 0.00%  126       // common sub-expressions
 0.00%  127
 0.00%  128       // evaluate aggregate function
 0.00%  129       agg_agg_isNull_2_0 = true;
 0.00%  130       double agg_value_2 = -1.0;
 0.00%  131       do {
 0.00%  132         boolean agg_isNull_3 = true;
 0.00%  133         double agg_value_3 = -1.0;
 0.00%  134         agg_agg_isNull_4_0 = true;
 0.10%  135         double agg_value_4 = -1.0;
 0.00%  136         do {
 0.00%  137           if (!agg_bufIsNull_0) {
 0.40%  138             agg_agg_isNull_4_0 = false;
```

```
0.00%  139              agg_value_4 = agg_bufValue_0;
0.00%  140              continue;
0.00%  141            }
0.00%  142
0.00%  143            boolean agg_isNull_6 = false;
0.00%  144            double agg_value_6 = -1.0;
0.00%  145            if (!false) {
0.00%  146              agg_value_6 = (double) 0;
0.00%  147            }
0.00%  148            if (!agg_isNull_6) {
0.00%  149              agg_agg_isNull_4_0 = false;
0.00%  150              agg_value_4 = agg_value_6;
0.00%  151              continue;
0.00%  152            }
0.00%  153
0.15%  154          } while (false);
0.00%  155
0.00%  156          if (!agg_exprIsNull_0_0) {
0.00%  157            agg_isNull_3 = false; // resultCode could change nullability.
0.00%  158            agg_value_3 = agg_value_4 + agg_expr_0_0;
0.00%  159
0.00%  160          }
0.04%  161          if (!agg_isNull_3) {
0.00%  162            agg_agg_isNull_2_0 = false;
0.17%  163            agg_value_2 = agg_value_3;
0.00%  164            continue;
0.00%  165          }
0.00%  166
0.00%  167          if (!agg_bufIsNull_0) {
0.00%  168            agg_agg_isNull_2_0 = false;
0.00%  169            agg_value_2 = agg_bufValue_0;
0.00%  170            continue;
0.00%  171          }
0.00%  172
0.06%  173        } while (false);
0.00%  174        // update aggregation buffer
0.00%  175        agg_bufIsNull_0 = agg_agg_isNull_2_0;
0.20%  176        agg_bufValue_0 = agg_value_2;
0.00%  177
0.00%  178      }
0.00%  179      /* /CodeRegion(CONSUME_97) */
0.00%  180      protected void processNext() throws java.io.IOException {
0.00%  181        /* CodeRegion(PRODUCE_97) */
0.00%  182        /* HashAggregate */
0.00%  183        while (!agg_initAgg_0) {
0.00%  184          agg_initAgg_0 = true;
0.00%  185          long agg_beforeAgg_0 = System.nanoTime();
0.00%  186          agg_doAggregateWithoutKey_0();
0.00%  187          ((SQLMetric) references[4] /* aggTime */).add((System.nanoTime() -
                                ↪ agg_beforeAgg_0) / 1000000);
0.00%  188
0.00%  189          // output the result
0.00%  190
0.00%  191          ((SQLMetric) references[3] /* numOutputRows */).add(1);
0.00%  192          /* CodeRegion(CONSUME_101) */
0.00%  193          /* WholeStageCodegen */
0.00%  194          scan_mutableStateArray_3[4].reset();
0.00%  195
0.00%  196          scan_mutableStateArray_3[4].zeroOutNullBytes();
0.00%  197
0.00%  198          if (agg_bufIsNull_0) {
0.00%  199          scan_mutableStateArray_3[4].setNullAt(0);
0.00%  200          } else {
0.00%  201          scan_mutableStateArray_3[4].write(0, agg_bufValue_0);
0.00%  202          }
0.00%  203          append((scan_mutableStateArray_3[4].getRow()));
0.00%  204
0.00%  205          /* /CodeRegion(CONSUME_101) */
0.00%  206        }
0.00%  207
0.00%  208        /* /CodeRegion(PRODUCE_97) */
0.00%  209      }
0.00%  210
0.00%  211    }
```