Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data

Grégory M. Essertel[1], Ruby Y. Tahboub[1], James M. Decker[1], Kevin J. Brown[2], Kunle Olukotun[2], Tiark Rompf[1]

[1]*Purdue University,* [2]*Stanford University*

{gesserte,rtahboub,decker31,tiark}@purdue.edu, {kjbrown,kunle}@stanford.edu

Abstract

In recent years, Apache Spark has become the de facto standard for big data processing. Spark has enabled a wide audience of users to process petabyte-scale workloads due to its flexibility and ease of use: users are able to mix SQL-style relational queries with Scala or Python code, and have the resultant programs distributed across an entire cluster, all without having to work with low-level parallelization or network primitives.

However, many workloads of practical importance are not large enough to justify distributed, scale-out execution, as the data may reside entirely in main memory of a single powerful server. Still, users want to use Spark for its familiar interface and tooling. In such scale-up scenarios, Spark's performance is suboptimal, as Spark prioritizes handling *data size* over optimizing the *computations* on that data. For such medium-size workloads, performance may still be of critical importance if jobs are computationally heavy, need to be run frequently on changing data, or interface with external libraries and systems (e.g., TensorFlow for machine learning).

We present Flare, an accelerator module for Spark that delivers order of magnitude speedups on scale-up architectures for a large class of applications. Inspired by query compilation techniques from main-memory database systems, Flare incorporates a code generation strategy designed to match the unique aspects of Spark and the characteristics of scale-up architectures, in particular processing data directly from optimized file formats and combining SQL-style relational processing with external frameworks such as TensorFlow.

1 Introduction

Modern data analytics applications require a combination of different programming paradigms, spanning relational, procedural, and map-reduce-style functional processing. Systems like Apache Spark [8] have gained enormous traction thanks to their intuitive APIs and ability to scale to very large data sizes, thereby commoditizing petabyte-scale (PB) data processing for large numbers of users. But thanks to its attractive programming interface and tooling, people are also increasingly using Spark for smaller workloads. Even for companies that *also* have PB-scale data, there is typically a long tail of tasks of much smaller size, which make up a very important class of workloads [17, 44]. In such cases, Spark's performance is suboptimal. For such medium-size workloads, performance may still be of critical importance if there are many such jobs, individual jobs are computationally heavy, or need to be run very frequently on changing data. This is the problem we address in this paper. We present Flare, an accelerator module for Spark that delivers order of magnitude speedups on scale-up architectures for a large class of applications. A high-level view of Flare's architecture can be seen in Figure 1b.

Inspiration from In-Memory Databases Flare is based on native code generation techniques that have been pioneered by in-memory databases (e.g., HyPer [35]). Given the multitude of front-end programming paradigms, it is not immediately clear that looking at relational databases is the right idea. However, we argue that this is indeed the right strategy: Despite the variety of front-end interfaces, contemporary Spark is, at its core, an SQL engine and query optimizer [8]. Rich front-end APIs are increasingly based on DataFrames, which are internally represented very much like SQL query plans. Data frames provide a deferred API, i.e., calls only *construct* a query plan, but do not execute it immediately. Thus, front-end abstractions do not interfere with query optimization. Previous generations of Spark relied critically on arbitrary UDFs, but this is becoming less and less of a concern as more and more func-

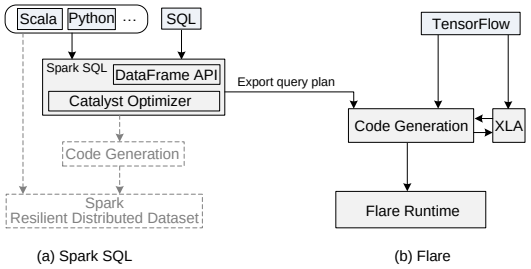


Figure 1: Flare system overview: (a) architecture of Spark adapted from [8]; (b) Flare generates code for entire queries, eliminating the RDD layer, and orchestrating parallel execution optimized for shared memory architectures. Flare also integrates with TensorFlow.

tionality is implemented on top of DataFrames.

With main-memory databases in mind, it follows that one may look to existing databases for answers on improving Spark's performance. A piece of low-hanging fruit seems to be simply translating all DataFrame query plans to an existing best-of-breed main-memory database (e.g., HyPer [35]). However, such systems are *full* database systems, not just query engines, and would require data to be stored in a separate, internal format specific to the external system. As data may be changing rapidly, loading this data into an external system is undesirable, for reasons of both storage size and due to the inherent overhead associated with data loading. Moreover, retaining the ability to interact with other systems (e.g., TensorFlow [3] for machine learning) is unclear.

Another logical alternative would be to build a new system which is overall better optimized than Spark for the particular use case of medium-size workloads and scale-up architectures. While some effort has been done in this vein (e.g., Tupleware [17]), such systems forfeit the ability to leverage existing libraries and frameworks built on top of Spark, including the associated tooling. Whereas a system that competes with Spark must replicate all of this functionality, our goal instead was to build a drop-in module capable of handling workloads for which Spark is not optimized, preferably using methodologies seen in these best-of-breed, external systems (e.g., HyPer).

Native Query Compilation Indeed, the need to accelerate CPU computation prompted the development of a code generation engine that ships with Spark since version 1.4, called Tungsten [8]. However, despite following some of the methodology set forth by HyPer, there are a number of challenges facing such a system, which causes Tungsten to yield suboptimal results by comparison. First, due to the fact that Spark resides in a Java-based ecosystem, Tungsten generates Java code. This (somewhat obviously) yields inferior performance to native execution as seen in HyPer. However, generating native code within Spark poses a challenge of interfacing with the JVM when dealing with e.g., data loading. Another challenge comes from Spark's reliance on resilient distributed datasets (RDDs) as its main internal execution abstraction. Mapping query operators to RDDs imposes boundaries between code generation regions, which incurs nontrivial runtime overhead. Finally, having a code generation engine capable of interfacing with external frameworks and libraries, particularly machine-learning oriented frameworks like TensorFlow and PyTorch, is also challenging due to the wide variety of data representations which may be used.

End-to-End Datapath Optimization In solving the problem of generating native code and working within the Java environment, we focus specifically on the issue of data processing. When working with data directly from memory, it is possible to use the Java Native Interface (JNI) and operate on raw pointers. However, when processing data directly from files, fine-grained interaction between decoding logic in Java and native code would be required, which is both cumbersome and presents high overhead. To resolve this problem, we elect to reimplement file processing for common formats in native code as well. This provides a fully compiled data path, which in turn provides significant performance benefits. While this does present a problem in calling Java UDFs (user-defined functions) at runtime, we can simply fall back to Spark's existing execution in such a case, as these instances appear rare in most use cases considered. We note in passing that existing work (e.g., Tupleware [17], Froid [40]) has presented other solutions for this problem which could be adopted within our method, as well.

Fault Tolerance on Scale-Up Architectures In addition, we must overcome the challenge of working with Spark's reliance on RDDs. For this, we propose a simple solution: when working in a scale-up, shared memory environment, remove RDDs and bypass all fault tolerance mechanisms, as they are not needed in such architectures (seen in Figure 1b). The presence of RDDs fundamentally limits the scope of query compilation to individual query stages, which prevents optimization at the granularity of full queries. Without RDDs, we compile whole queries and eliminate the preexisting boundaries across query stages. This also enables the removal of artifacts of distributed architectures, such as Spark's use of `HashJoinExchange` operators even if the query is run on a single core.

Interfacing with External Code Looking now to the issue of having a robust code generation engine capable of interfacing with external libraries and frameworks within Spark, we note that most performance-critical external frameworks are *also* embracing deferred APIs. This is particularly true for machine learning frameworks, which are based on a notion of execution graphs. This includes popular frameworks like TensorFlow [3], Caffe [24], and ONNX [1], though this list is far from exhaustive. As such, we focus on frameworks with APIs that follow this pattern. Importantly, many of these systems already have a native execution backend, which allows for speedups by generating all required glue code and keeping the entire data path within native code.

Contributions The main intellectual contribution of this paper is to demonstrate and analyze some of the underlying issues contained in the Spark runtime, and to show that the HyPer query compilation model must be adapted in certain ways to achieve good results in Spark (and, most likely, systems with a similar architecture like Flink [16]), most importantly to eliminate codegen boundaries as much as possible. For Spark, this means generating code not at the granularity of operator pipelines but compiling whole Catalyst operator trees at once (which may include multiple SQL-queries and subqueries), generating specialized code for data structures, for file loading, etc.

We present Flare, an accelerator module for Spark that solves these (and other) challenges which currently prevent Spark from achieving optimal performance on scale-up architectures for a large class of applications. Building on query compilation techniques from main-memory database systems, Flare incorporates a code generation strategy designed to match the unique aspects of Spark and the characteristics of scale-up architectures, in particular processing data directly from optimized file formats and combining SQL-style relational processing with external libraries such as TensorFlow.

This paper makes the following specific contributions:

- We identify key impediments to performance for medium-sized workloads running on Spark on a single machine in a shared memory environment and present a novel code generation strategy able to overcome these impediments, including the overhead inherent in boundaries between compilation regions. (Section 2).
- We present Flare's architecture and discuss some implementation choices. We show how Flare is capable of optimizing data loading, dealing with parallel execution, as well as efficiently working on NUMA systems. This is a result of Flare compiling whole queries, as opposed to individual query stages, which results in an end-to-end optimized data path (Section 3).
- We show how Flare's compilation model efficiently extends to external user-defined functions. Specifically, we discuss Flare's ability to integrate with other frameworks and domain-specific languages, including in particular machine learning frameworks like TensorFlow that provide compilation facilities of their own (Section 4).
- We evaluate Flare in comparison to Spark on TPC-H, reducing the gap to best-of-breed relational query engine, and on benchmarks involving external libraries. In both settings, Flare exhibits order-of-magnitude speedups. Our evaluation spans single-core, multi-core, and NUMA targets (Section 5).

Finally, we survey related work in Section 6, and draw conclusions in Section 7.

2 Background

Apache Spark [55, 56] is today's most widely-used big data framework. The core programming abstraction comes in the form of an immutable, implicitly distributed collection called a resilient distributed dataset (RDD). RDDs serve as high-level programming interfaces, while also transparently managing fault-tolerance.

We present a short example using RDDs (from [8]), which counts the number of errors in a (potentially distributed) log file:

```
val lines  = spark.sparkContext.textFile("...")
val errors = lines.filter(s => s.startsWith("ERROR"))
println("Total errors: " + errors.count())
```

Spark's RDD abstraction provides a *deferred* API: in the above example, the calls to `textFile` and `filter` merely construct a computation graph. In fact, no actual computation occurs until `errors.count` is invoked.

The directed, acyclic computation graph represented by an RDD describes the distributed operations in a rather coarse-grained fashion: at the granularity of `map`, `filter`, etc. While this level of detail is enough to enable demand-driven computation, scheduling, and fault-tolerance via selective recomputation along the "lineage" of a result [55], it does not provide a full view of the computation applied to each element of a dataset. For example, in the code snippet shown above, the argument to `lines.filter` is a normal Scala closure. This makes integration between RDDs and arbitrary external libraries much easier, but it also means that the given closure must be invoked as-is for every element in the dataset.

As such, the performance of RDDs suffers from two limitations: first, limited visibility for analysis and optimization (especially standard optimizations, e.g., join

reordering for relational workloads); and second, substantial interpretive overhead, i.e., function calls for each processed tuple. Both issues have been ameliorated with the introduction of the Spark SQL subsystem [8].

2.1 The Power of Multi-Stage APIs

The chief addition of Spark SQL is an alternative API based on DataFrames. A DataFrame is conceptually equivalent to a table in a relational database; i.e., a collection of rows with named columns. However, like RDDs, the DataFrame API records operations, rather than computing the result.

Therefore, we can write the same example as before:

```
val lines  = spark.read.textFile("...")
val errors = lines.filter($"value".startsWith("ERROR"))
println("Total errors: " + errors.count())
```

Indeed, this is quite similar to the RDD API in that only the call to `errors.count` will trigger actual execution. Unlike RDDs, however, DataFrames capture the *full* computation/query to be executed. We can obtain the internal representation using `errors.explain()`, which produces the following output:

```
== Physical Plan ==
*Filter StartsWith(value#894, ERROR)
+- *Scan text [value#894]
     Format: ...TextFileFormat@18edbdbb,
     InputPaths: ...,
     ReadSchema: struct<value:string>
```

From the high-level DataFrame operations, Spark SQL computes a *query plan*, much like a relational DBMS. Spark SQL optimizes query plans using its relational query optimizer, called Catalyst, and may even generate Java code at runtime to accelerate parts of the query plan using a component named Tungsten (see Section 2.2).

It is hard to overstate the benefits of this kind of deferred API, which generates a complete program (i.e., query) representation at runtime. First, it enables various kinds of optimizations, including classic relational query optimizations. Second, one can use this API from multiple front-ends, which exposes Spark to non-JVM languages such as Python and R, and the API can also serve as a translation target from literal SQL:

```
lines.createOrReplaceTempView("lines")
val errors = spark.sql("select * from lines
                        where value like 'ERROR%'")
println("Total errors: " + errors.count())
```

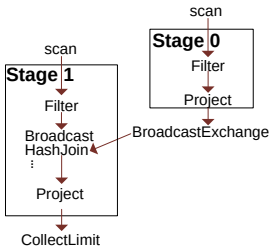
Third, one can use the full host language to structure code, and use small functions that pass DataFrames between them to build up a logical plan that is then optimized as a whole.

However, this is only true as long as one stays in the relational world, and, notably, avoids using any external libraries (e.g., TensorFlow). This is a nontrivial restriction; to resolve this, we show in Section 4 how the DataFrame model extends to such library calls in Flare.

```
select *
from lineitem, orders
where l_orderkey = o_orderkey

Spark
  Sort-merge join         14,937 ms
  Broadcast-hash join      4,775 ms
    of which in exchange   2,232 ms
Flare
  In-memory hash join       136 ms
```

(a)



(b)

Figure 2: (a) The cost of Join `lineitem` ⋈ `orders` with different operators (b) Spark's hash join plan shows two separate code generation regions, which communicate through Spark's runtime system.

2.2 Catalyst and Tungsten

With the addition of Spark SQL, Spark also introduced a query optimizer known as Catalyst [8]. We elide the details of Catalyst's optimization strategy, as they are largely irrelevant here. After Catalyst has finished optimizing a query plan, Spark's execution backend known as Tungsten takes over. Tungsten aims to improve Spark's performance by reducing the allocation of objects on the Java Virtual Machine (JVM) heap, controlling off-heap memory management, employing cache-aware data structures, and generating Java code which is then compiled to JVM bytecode at runtime [28]. Notably, these optimizations are able to simultaneously improve the performance of *all* Spark SQL libraries and DataFrame operations [56].

Following the design described by Neumann, and implemented in HyPer [35], Tungsten's code generation engine implements what is known as a "data-centric" model. In this type of model, operator interfaces consist of two methods: `produce`, and `consume`. The `produce` method on an operator signals all child operators to begin producing data in the order defined by the parent operator's semantics. The `consume` method waits to receive and process this data, again in accordance with the parent operator's semantics.

In HyPer (and Tungsten), operators that materialize data (e.g., aggregate, hash join, etc.) are called "pipeline breakers". Where possible, pipelines of operators (e.g., scan, aggregate) are fused to eliminate unnecessary function calls which would otherwise move data between operators. A consequence of this is that all code generated is at the granularity of query *stage*, rather than generating code for the query as a whole. This requires some amount of "glue code" to also be generated, in order to pipeline these generated stages together. The directed graph of the physical plan for a simple join query can be seen in Figure 2b. In this figure, we can see that the first stage generates code for scanning and filtering the first

```
val tpchq6 = spark.sql("""
  select
    sum(l_extendedprice*l_discount) as
          revenue
  from
    lineitem
  where
    l_shipdate >= to_date('1994-01-01')
    and l_shipdate < to_date('1995-01-01')
    and l_discount between 0.05 and 0.07
    and l_quantity < 24
""")
```

(a)

```
// data loading elided ...
for (i = 0; i < size; i++) {
  double l_quantity = l_quantity_col[i];
  double l_extendedprice = l_extendedprice_col[i];
  double l_discount = l_discount_col[i];
  long l_shipdate = l_shipdate_col[i];
  if (l_shipdate >= 19940101L && l_shipdate <
      19950101L &&
      l_discount >= 0.05 && l_discount <= 0.07 &&
      l_quantity < 24.0) {
    revenue += l_extendedprice * l_discount;
  }
} ...
```

(b)

Spark SQL	Preload ms	Query ms
Direct CSV	-	24,400
Preload CSV	118,062	1,418
Hand-Written C / Flare		
Preload CSV	2,847	45

(c)

Figure 3: (a) Query 6 from the TPC-H benchmark in Spark (b) Q6 hand-written C code (c) Running times for Q6 in Spark, with and without pre-loading, and compared to hand-written code and Flare.

table and the second stage generates code for the pipeline of the scan, join, and project operators. In Section 2.4 we discuss the impact of the granularity of code generation and the choice of join algorithm on Spark's performance.

2.3 Spark Performance Analysis

Spark performance studies primarily focus on the scale-out performance, e.g., running big data benchmarks [55] on high-end clusters, performing terabyte sorting [56], etc. However, when considering the class of computationally-heavy workloads that can fit in main-memory, requires multiple iterations, or integrates with external libraries (e.g., training a machine learning classifier), the performance of Spark becomes suboptimal.

On a similar note, McSherry, Isard, and Murray have eloquently argued in their 2015 HotOS paper [30] and accompanying blog post [29] that big data systems such as Spark tend to scale well, but often this is because there is a lot of internal overhead. In particular, McSherry et al. demonstrate that a straightforward native implementation of the PageRank algorithm [37] running on a single laptop can outperform a Spark cluster with 128 cores, using the then-current version.

Laptop vs. Cluster Inspired by this setup and the following quote, we are interested in gauging the inherent overheads of Spark and Spark SQL in absolute terms:

> "You can have a second computer once you've shown you know how to use the first one."
>
> — Paul Barham, via [30]

For our benchmark, we pick the simplest query from the industry-standard TPC-H benchmark: Query 6 (shown in Figure 3a). We define the schema of table `lineitem`, provide the source file, and finally register it as a temporary table for Spark SQL (steps not shown). For our experiments, we use scale factor 2 (SF2) of the TPC-H data set, which means that table `lineitem` is stored in a CSV file of about 1.4 GB. Following the setup by McSherry et al., we run our tests on a fairly standard laptop.[1] All times referenced below may be found in Figure 3c.

We first do a naive run of our query, Q6. As reported in Figure 3, we achieve a result of 24 seconds, which is clearly suboptimal. In aiming to boost performance, one option at this is to convert our data to the columnar Parquet format [7] for increased performance. Alternatively, we can preload the data so that subsequent runs are purely in-memory. As we are mainly interested in the computational part, we opt to preload.

We note in passing that preloading is quite slow (almost 2 min), which may be due to a variety of factors. With things preloaded, however, we can now execute our query in-memory, and we get a much better result of around 1.4 seconds. Running the query a few more times yields further speedups, but timings stagnate at around 1 second (timing from subsequent runs elided). Using 1s as our baseline, we must now qualify this result.

Hand-Written C Due to the simplicity of Q6, we elect to write a program in C which performs precisely the same computation: mapping the input file into memory using the `mmap` system call, loading the data into an in-memory columnar representation, and then executing the main query loop (see Figure 3b).

Compiling this C program via `gcc -O3 Q6.c` and running the resultant output file yields a time of 2.8 seconds (including data loading), only 45ms of which is performing the actual query computation. Note that in comparison to Spark 2.0, this is a striking 20× speedup. Performing the same query in HyPer, however, takes only 46.58ms, well within the margin of error of the hand-written C code. This disparity in performance shows that although Tungsten is written with the methodologies prescribed by HyPer in mind, there exist some impediments either in the implementation of these methodologies or in the Spark runtime itself which prevent Spark from achieving optimal performance for these cases.

[1] MacBook Pro Retina 2012, 2.6 GHz Intel Core i7, 16 GB 1600 MHz DDR3, 500 GB SSD, Spark 2.0, Java HotSpot VM 1.8.0_112-b16

```
case class BroadcastHashJoinExec(/* ... inputs elided ... */)
  extends BinaryExecNode with HashJoin with CodegenSupport {
  // ... fields elided ...
  override def doProduce(ctx: CodegenContext): String =
    streamedPlan.asInstanceOf[CodegenSupport].produce(ctx, this)
  override def doConsume(ctx: CodegenContext, input:
        Seq[ExprCode], row: ExprCode): String = {
    val (broadcastRelation, relationTerm) = prepareBroadcast(ctx)
    val (keyEv, anyNull) = genStreamSideJoinKey(ctx, input)
    val (matched, checkCondition, buildVars) =
          getJoinCondition(ctx, input)
    val numOutput = metricTerm(ctx, "numOutputRows")

    val resultVars = ...
    ctx.copyResult = true
    val matches = ctx.freshName("matches")
    val iteratorCls = classOf[Iterator[UnsafeRow]].getName
    s"""
      |// generate join key for stream side
      |${keyEv.code}
      |// find matches from HashRelation
      |$iteratorCls $matches = $anyNull ? null :
            ($iteratorCls)$relationTerm.get(${keyEv.value});
      |if ($matches == null) continue;
      |while ($matches.hasNext()) {
      |  UnsafeRow $matched = (UnsafeRow) $matches.next();
      |  $checkCondition
      |  $numOutput.add(1);
      |  ${consume(ctx, resultVars)}
      |}
    """.stripMargin
  }
}
```

Figure 4: Spark implementation of inner HashJoin.

2.4 Major Bottlenecks

By profiling Spark SQL during a run of Q6, we are able to determine two key reasons for the large gap in performance between Spark and HyPer. Note that while we focus our discussion mainly on Q6, which requires low computational power and uses only trivial query operators, these bottlenecks appear in nearly every query in the TPC-H benchmark.

Data Exchange Between Code Boundaries We first observe that Tungsten must generate multiple pieces of code: one for the main query loop, the other an iterator to traverse the in-memory data structure.

Consider the HashJoin code in Figure 4. We can see that Tungsten's produce/consume interface generates a loop which iterates over data through an iterator interface, then invokes the `consume` method at the end of the loop in order to perform evaluation. HyPer's original codegen model is centrally designed around data-centric pipelines within a given query, the notion of "pipeline-breakers" as coarse-grained boundaries of data flow, and the combination of pre-written code at the boundary between pipelines with generated code within each pipeline. While the particular implementation of this design in HyPer leads to good results in HyPer itself, the direct implementation of HyPer's pipeline-focused approach in Spark and similar systems falls short because the overhead of traversing pipeline boundaries is much higher (Java vs C++, RDD overhead, ecosystem integration, etc).

The CPU profile (Figure 5) shows that 80% of the execution time is spent in one of two ways: accessing and decoding the in-memory data representation, or moving between the two pieces of generated code through code paths which are part of the precompiled Spark runtime. In order to avoid this overhead, then, we must replace the runtime altogether with one able to reason about the *entire* query, rather than just the stages.

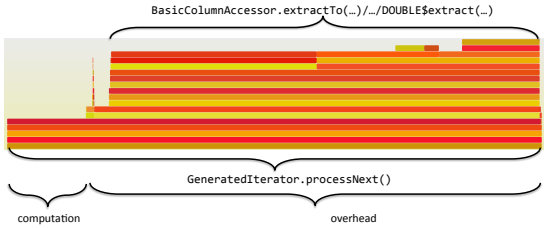


Figure 5: CPU profile of TPC-H Q6 in Spark SQL, after preloading the `lineitem` table. 80% of time is spent accessing and decoding the in-memory data representation.

JVM Overhead Even if the previous indirection is removed and replaced with a unified piece of Java code, the performance remains approximately 30% lower than our hand-written C code. This difference becomes more pronounced in other TPC-H queries which require both memory management and tighter low-level control over data structures. This bottleneck is certainly expected, and choosing a lower level language does alleviate this performance loss greatly.

Other Bottlenecks As shown, even fixing these bottlenecks is not enough. This becomes even more apparent when moving away from Q6. In dealing with more complex queries, concerns regarding granularity of code generation and the necessity to interface with the Spark runtime system become more pronounced than with TPC-H Q6. In fact, queries which require join operations exhibit some unfortunate consequences for main-memory execution due to Spark's design as primarily a cluster-computing framework. Figure 2a shows timings for a simple join query that joins the `lineitem` and `orders` tables of the TPC-H benchmark. Spark's query optimizer picks an expensive sort-merge join by default. Note that this may be the correct choice for distributed or out-of-core execution, but is suboptimal for main memory. With some tuning, it is possible to force Spark's query planner to opt for a hash join instead, which is more efficient for our architecture. However, even this follows a broadcast model with high overhead for the internal exchange operator (2.2s of 4.7s) which is present in the physical plan even when running on a single core.

3 Flare: Adding Fuel to the Fire

Based on the observations made in Sections 2.3 and 2.4, we formally present Flare: a new backend which acts as an accelerator for Spark for medium-sized workloads

on scale-up architectures. Flare eliminates all previously identified bottlenecks *without* removing the expressiveness and power of its front-ends. At its core, Flare efficiently generates code, and brings Spark's performance closer to HyPer and hand-written C. Flare compiles whole queries instead of only query *stages*, effectively bypassing Spark's RDD layer and runtime for operations like hash joins in shared-memory environments. Flare also goes beyond purely relational workloads by adding another intermediate layer between query plans and generated code.

As previously identified, this need to build a new runtime, rather than selecting an existing system as an alternate backend for Spark, is founded on a number of justifications. In particular, we focus on the deferred API provided by Spark SQL which builds computation graphs to perform the necessary queries as given by users. Access to this graph structure allows for cross-optimization with external libraries also using deferred APIs (e.g., TensorFlow) through the use of robust code generation techniques. In order to gain access to the necessary data at the appropriate time without incurring the overhead of passing (potentially large amounts of) data between external programs, a new runtime capable of interfacing with Spark's existing front-end is required.

3.1 Interface between Spark and Flare

Flare supports most available Spark SQL DataFrame or DataSet operations (i.e., all operations which can be applied to a DataFrame and have a representation as Catalyst operators), though any operators currently missing could be added without compatibility constraints. In the event that a Spark job contains operations that are not part of the SQL frontend, Flare can still be used to accelerate SQL operations and then return the result to the Spark runtime, which will then use the result for the rest of the computation. However, the benefit of doing this may be negated by the communication overhead between the two systems.

Flare can operate in one of two modes. Either users must invoke a function to convert the DataFrame they wish to compute into a Flare DataFrame (a conversion that may fail with a descriptive error), to that end Flare exposes a dedicated API to allow users to pick which DataFrames to evaluate through Flare:

```
val df = spark.sql("...") // create DataFrame (SQL or direct)
val fd = flare(df)        // turn it into a FlareDataFrame
fd.show()                 // execute query plan with Flare
```

Or one can set a configuration item in Spark to use Flare on all queries where possible, and only fall back to the default Spark execution when necessary (optionally emitting a warning when doing so). When Flare is

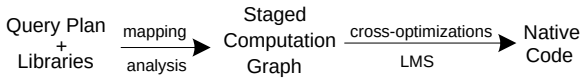


Figure 6: Query compilation in Flare

invoked, it first generates C code as explained in the following section, then invokes a C compiler, and finally launches the resulting binary either inside the JVM, or as a separate process. This bypasses Spark's runtime entirely, relying solely on Flare's runtime to trigger execution of the generated code.

3.2 Flare: Architecture

A high-level view of Flare's architecture is illustrated in Figure 1b. Spark SQL's front-end, the DataFrame API, and the Catalyst optimizer all remain the same. When dealing with relational workloads, the optimized query plan is exported without modification from Catalyst to Flare, upon which Flare performs a compilation pass and creates a code generation object for each of the nodes.

At a closer look, Figure 6 illustrates an end-to-end execution path in Flare. Flare analyzes Spark's optimized plan (which possibly embeds external libraries as UDFs) and constructs a computation graph that encodes relational operators, data structures, UDFs, data layout, and any needed configurations.

Generative Programming and Lightweight Modular Staging (LMS) Flare uses Lightweight Modular Staging (LMS) for code generation due to its multi-staging capabilities. In LMS, a special type constructor `Rep[T]` is used to denote a *staged* expression, which will cause an expression of type `T` to be emitted in the generated code. The example in Figure 7a shows a code snippet that generates a for loop (notice the loop counter is specialized with `Rep`). At evaluation time, the for loop iteration will be generated. On the other hand, the `if` condition is composed of a regular Boolean type, so this code is executed at code generation time as shown in Figure 7b.

```
val squared: Boolean = true
val arr = NewArray[Int](5)
for (i <- 0 until 5: Rep[Range]) {
  if (squared)
    arr(i) = i * i
  else
    arr(i) = i
}
```

(a)

```
int* x1 = malloc(5 *
     sizeof(int));
int x2;
for (x2 = 0; x2 < 5; x2++) {
 int x3 = x2 * x2;
 x1[x2] = x3;
}
```

(b)

Figure 7: (a) Generic LMS code example (only used for Flare internals) (b) LMS generated C code.

The essence of multi-stage programming is to generate efficient programs using high-level constructs without incurring runtime overheads [47]. In the late 1990s, it was realized that multi-stage languages (i.e., languages used

to express multi-stage programs) are useful not only as intermediate formal description, but also directly as programming tools, giving rise to the field of programmable specialization or *generative programming*, embodied by languages like MetaML [48] and MetaOCaml [15]. Even more recently, library-based approaches have become popular that implement generative programming abstractions using operator overloading and other features in regular general-purpose languages. One instance of such a system is LMS [43], implemented in Scala. LMS maintains a graph-like intermediate representation (IR) to encode constructs and operations. LMS introduces a type constructor called `Rep[T]` (where T is a type, e.g., `String`) to denote expressions that will generate code. For example, given two `Rep[Int]` values `a` and `b`, *evaluating* the expression `a+b` will *generate* the following code: `int x1 = a + b` and return a reference to `x1` as new `Rep[Int]` result in the meta language. This value can then be used in other computations.

Staging and code generation in Flare In the context of query compilation, LMS is used to specialize a query evaluator with respect to a query plan [42, 25]. Based on partial evaluation results (the first Futamura projection [21]), the outcome of programmatic specialization is a compiled target of the query. Figure 8 shows an example of compiling a join query in Flare, in which the specialization logic (i.e., staging code using `Rep`) is placed at the granularity of low-level control flow constructs and primitive operators.

```
case class DataLoop(foreach: (Rep[Record] => Unit) => Unit)
type ThreadCallback = Rep[Int] => DataLoop => Unit
case class CodeGen(gen: ThreadCallback => Unit)

// extract the join hash key functions
def compileCond(cond: Option[Expression]): (Rep[Record] =>
      Rep[Record], Rep[Record] => Rep[Record]) = ...
def compile(plan: LogicalPlan): CodeGen = plan match {
  case HashJoin(left, right, Inner, cond) =>
    val lGen = compile(left); val rGen = compile(right)
    val (lkey, rkey) = compileCond(cond)

    val hmap = new ParHashMap[Record,Record]()
    CodeGen(threadCallback =>
      lGen.gen { tId => dataloop => // start section for left child
        val lhmap = hmap.partition(tId) // Thread local data
        for (ltuple <- dataloop) lhmap += (lkey(ltuple), ltuple)
      }
      rGen.gen { tId => dataloop => // start section for right
            child
        threadCallback(tId) { callback => // invoke downstream op
          for (rtuple <- dataloop)
            for (ltuple <- hmap(rkey(rtuple)))
              callback(merge(ltuple, rtuple)) // feed downstream op
    } } )
  case ...
}
```

Figure 8: Internal Flare operator that generates code for HashJoin (LogicalPlan and HashJoin are Spark classes).

Following the `InnerJoin` code generation example in Figure 8, a `CodeGen` object is generated from each of the two children, after which the logic of the `Join` operator is implemented: the left child's code generator is invoked and the tuples produced populate a hash map. The right child's code generator is then invoked, and for each of the tuples produced, the matching lines from the left table are extracted from the map, merged, and finally become the produced value of the `Join` operator. LMS performs some lightweight optimizations (e.g., common subexpression elimination, dead code elimination), and generates C code that can be compiled and executed by the Flare runtime.

Interestingly, this implementation looks exactly like the implementation of an interpreter. Indeed, this is no coincidence: much like Spark uses multi-stage APIs (Section 2.1) at the operational level, Flare uses the LMS compiler framework, which implements the same concept, but at a lower level. In the same way that Scala (or Python) is used to build DataFrames in Spark, we use Scala to build a graph which represents the computations needing to be generated. We qualify the code generation of Spark as coarse-grain. The `BroadcastHashJoinExec` operator in Figure 4 generates a string that corresponds to the full join computation. This String is generated with regard to some placeholders for the inputs/outputs and join conditons that are specific to the given query. However, what is hardcoded in the template string will be generated in the same way for every join. Contrast this with Flare's fine-grained code generation: The code in Figure 8 also generates code for the Join operator. However, it does not generate one big string; rather, it invokes functions that express the logic of the operator using the full power of the Scala language. The use of `Rep[T]` expressions in judicious places triggers code generation and produces only low-level operations.

With the goal of removing the tightest bottlenecks first, the implementation of Flare has focused on maximizing performance within a single machine. Therefore, Flare does not implement any specific optimizations for distributed execution. Furthermore, Flare is also unable to handle any workloads which require more memory than the machine has available. In either of these cases, we fall back to the Spark runtime.

3.3 Optimizing Data Loading

Data loading is an often overlooked factor data processing, and is seldom reported in benchmarks. However, we recognize that data loading from CSV can often be the dominant performance factor for Spark SQL queries. The Apache Parquet [7] format is an attractive alternative, modeled after Dremel [31]. As a binary columnar format, it offers opportunities for compression, and queries can load only required columns.

While Parquet allows for irrelevant data to be ignored almost entirely, Spark's code to read Parquet files is very generic, resulting in undue overhead. This generality is

primarily due to supporting multiple compression and encoding techniques, but there also exists overhead in determining which column iterators are needed. While these sources of overhead seem somewhat unavoidable, in reality they can be resolved by generating specialized code. In Flare, we implement compiled CSV and Parquet readers that generate native code specialized to a given schema. As a result, Flare can compile data paths end-to-end. We evaluate these readers in Section 5.

3.4 Indexing Structures

Query engines build indexing structures to minimize time spent in table lookups to speed-up query execution. Small-size data processing is performed efficiently using table scans, whereas very large datasets are executed in latency-insensitive contexts. On the other hand, medium-size workloads can profit from indexes, as these datasets are often processed under tight latency constraints where performing full table scans is infeasible. On that basis, Flare supports indexing structures on primary and foreign keys. At the time of writing, Spark SQL does not support index-based plans. Thus, Flare adds metadata to the table schema that describes index type and key attributes. At loading time, Flare builds indexes as specified in the table definition. Furthermore, Flare implements a set of index-based operators, e.g., scan and join following the methodology described in [49]. Finally, at compilation time, Flare maps Spark's operators to use the index-based operators if such an index is present. The index-based operators are implemented with the same technique described for the basic operators, but shortcut some computation by using the index rather than requesting data from its children.

3.5 Parallel and NUMA Execution

Query engines can implement parallelism either explicitly through special *split* and *merge* operators, or internally by modifying the operator's internal logic to orchestrate parallel execution. Flare does the latter, and currently realizes parallelism using OpenMP [2] annotations within the generated C code, although alternatives are possible. On the architectural level, Flare handles splitting the computation internally across multiple threads, accumulating final results, etc. For instance, the parallel `scan` starts a parallel section, which sets the number of threads and invokes the downstream operators in parallel through a `ThreadCallback` (see Figure 8). `join` and `aggregate` operators, in turn, which implement materialization points, implement their `ThreadCallback` method in such a way that parallel invocations are possible without conflict. This is typically accomplished through either per-thread data structures that are merged after the parallel section or lock-free data structures.

Flare also contains specific optimizations for environments with non-uniform memory access (NUMA), including pinning threads to specific cores and optimizing the memory layout of various data structures to reduce the need for accessing non-local memory. For instance, memory-bound workloads (e.g., TPC-H Q6) perform small amounts of computation, and do not scale up given a large number of threads on a single CPU socket. Flare's code generation supports such workloads through various data partitioning strategies in order to maximize local processing and to reduce the need for threads to access non-local memory as illustrated Section 5.1.

4 Heterogeneous Workloads

Many data analytics applications require a combination of different programming paradigms, e.g., relational, procedural, and map-reduce-style functional processing. For example, a machine learning (ML) application might use relational APIs for the extract, transform, load phase (ETL), and dedicated ML libraries for computations. Spark provides specialized libraries (e.g., ML pipelines), and supports user-defined functions to support domain-specific applications. Unfortunately, Spark's performance is greatly diminished once DataFrame operations are interleaved with calls to external libraries. Currently, Spark SQL optimization and code generation treat calls to such libraries as calls to black boxes. Hence, Flare focuses on generating efficient code for heterogeneous workloads including external systems e.g., TensorFlow [4].

4.1 User Defined Functions (UDF)

Spark SQL uses Scala functions, which appear as a black box to the optimizer. As mentioned in Section 3.2, Flare's internal code generation logic is based on LMS, which allows for multi-stage programming using `Rep` types. Extending UDF support to Flare is achieved by injecting `Rep[A] => Rep[B]` functions into DataFrames in the same way that normal `A => B` functions are injected in plain Spark. As an example, here is a UDF `sqr` that squares a given number:

```
// define and register UDF
def sqr(fc: FlareUDFContext) = { import fc._;
    (y: Rep[Int]) => y * y }
flare.udf.register("sqr", sqr)
// use UDF in query
val df = spark.sql("select ps_availqty from partsupp where
                    sqr(ps_availqty) > 100")
flare(df).show()
```

Notice that the definition of `sqr` uses an additional argument of type `FlareUDFContext`, from which we import overloaded operators such as `+`, `-`, `*`, etc., to work on `Rep[Int]` and other `Rep[T]` types. The staged function will become part of the code as well, and will be

```
# Define linear classifier using TensorFlow
import tensorflow as tf
# weights from pre-trained model elided
mat = tf.constant([[...]])
bias = tf.constant([...])
def classifier(c1,c2,c3,c4):
  # compute distance
  x = tf.constant([[c1,c2,c3,c4]])
  y = tf.matmul(x, mat) + bias
  y1 = tf.session.run(y1)[0]
  return max(y1)
# Register classifier as UDF: dumps TensorFlow graph to
# a .pbtxt file, runs tf_compile to obtain .o binary file
flare.udf.register_tfcompile("classifier", classifier)
# Use compiled classifer in PySpark query with Flare:
q = spark.sql("
  select real_class,
    sum(case when class = 0 then 1 else 0 end) as class1,
    sum(case when class = 1 then 1 else 0 end) as class2,
    ... until 4 ...
  from (select real_class,
               classifier(c1,c2,c3,c4) as class from data)
  group by real_class order by real_class")
flare(q).show()
```

Figure 9: Spark query using TensorFlow classifier as a UDF in Python.

optimized along with the relational operations. This provides benefits for UDFs (general purpose code embedded in queries), and enables queries to be be optimized with respect to their surrounding code (e.g., queries run within a loop).

4.2 Native UDF Example: TensorFlow

Flare has the potential to provide significant performance gains with other machine learning frameworks that generate native code. Figure 9 shows a PySpark SQL query which uses a UDF implemented in TensorFlow [3, 4]. This UDF performs classification via machine learning over the data, based on a pretrained model. It is important to reiterate that this UDF is seen as a black box by Spark, though in this case, it is also opaque to Flare.

Calling TensorFlow code from Spark hits a number of bottlenecks, resulting in poor performance (see Section 5). This is in large part due to the separate nature of the two programs; there is no inherent way to "share" data without copying back and forth. A somewhat immediate solution is to use the JNI, which enables the use of TensorFlow's ahead-of-time (AOT) compiler, XLA [50]. This already improves performance by over 100×, but even here there is room for improvement.

Using Flare in conjunction with TensorFlow provides speedups of over 1,000,000× when compared with Spark (for concrete numbers, see Section 5). These gains come primarily as a result of Flare's ability to link with external C libraries. As mentioned previously, in this example, Flare is able to take advantage of XLA, whereas Spark is relegated to using TensorFlow's less efficient dynamic runtime (which executes a TensorFlow computation graph with only limited knowledge). Flare provides a function `flare.udf.register_tfcompile`, which internally creates a TensorFlow subgraph representing the UDF, saves it to a file, and then invokes TensorFlow's AOT compiler tool `tfcompile` to obtain a compiled object file, which can then be linked against the query code generated by Flare.

Finally, the TensorFlow UDF generated by XLA is pure code, i.e., it does not allocate its own memory. Instead, the caller needs to preallocate all memory which will be used by the UDF. Due to its ability to generate native code, Flare can organize its own data structures to meet TensorFlow's data requirements, and thus does not require data layout modification or extraneous copies.

5 Experimental Evaluation

To assess the performance and acceleration potential of Flare in comparison to Spark, we present two sets of experiments. The first set focuses on a standard relational benchmark; the second set evaluates heterogeneous workloads, consisting of relational processing combined with a TensorFlow machine learning kernel. Our experiments span single-core, multi-core, and NUMA targets.

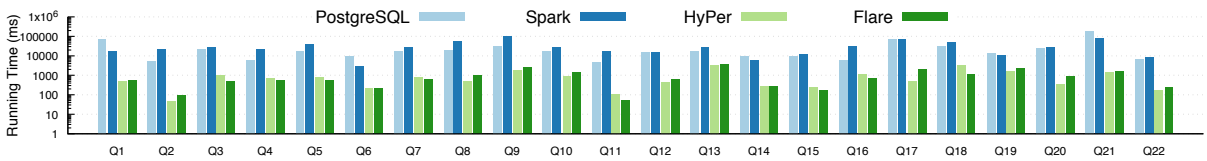
5.1 Bare-Metal Relational Workloads

The first set of experiments focuses on a standard relational workload, and demonstrates that the inherent overheads of Spark SQL cause a slowdown of at least 10× compared to the best available query engines for in-memory execution on a single core. Our experiments show that Flare is able to bridge this gap, accelerating Spark SQL to the same level of performance as state-of-the-art query compiler systems, while retaining the flexibility of Spark's DataFrame API. We also compare parallel speedups, the effect of NUMA optimization, and evaluate the performance benefits of optimized data loading.

Environment We conducted our experiments on a single NUMA machine with 4 sockets, 24 Xeon(R) Platinum 8168 cores per socket, and 750GB RAM per socket (3 TB total). The operating system is Ubuntu 16.04.4 LTS. We use Spark 2.3, Scala 2.11, Postgres 10.2, HyPer v0.5-222-g04766a1, and GCC 5.4 with optimization flags `-O3`.

Dataset We use the standard TPC-H [51] benchmark with scale factor SF10 for sequential, and SF20 and SF100 for parallel execution.

Single-Core Running Time In this experiment, we compare the single-core, absolute running time of Flare with Postgres, HyPer, and Spark using the TPC-H benchmark with scale factor SF10. In the case of Spark, we use a single executor thread, though the JVM may spawn auxiliary threads to handle GC or the just-in-time compilation. Postgres and HyPer implement cost-based optimizers that can avoid inefficient query plans, in partic-



SF10	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Postgres	72599	5064	20790	6032	17904	9071	16408	20003	31728	16409	4729	15755	16152	9369	9607	5662	66254	30104	12852	22795	184736	6900
Spark SQL	17692	21062	28039	20693	40008	2826	28262	55346	98216	27585	16193	15431	25632	5913	11538	30973	66089	47542	11062	27448	78896	8687
HyPer	498	47	969	725	821	207	804	467	1782	846	111	460	3307	283	227	1112	485	3061	1666	345	1349	177
Flare	550	98	481	544	527	212	596	1016	2688	1429	53	643	3499	256	175	679	1924	1150	2236	836	1533	241

Figure 10: Performance comparison of Postgres, HyPer, Spark SQL, Flare in SF10

ular by reordering joins. While Spark's Catalyst optimizer [8] is also cost-based, the default configurations do not perform any kind of join re-ordering. Hence, we match the join ordering of the query plan in Spark SQL and Flare with HyPer's, with a small number of exceptions: in Spark SQL, the original join ordering given in the TPC-H reference outperformed the HyPer plans for Q5, Q9, Q10, and Q11 in Spark SQL, and for Q10 in Flare. For these queries, we kept the original join ordering as is. For Spark SQL, this difference is mainly due to Catalyst picking sort-merge joins over hash joins. It is worth pointing out that HyPer and Postgres plans can use indexes on primary keys, which may give an additional advantage.

Figure 10 gives the absolute execution time of Postgres, HyPer, Spark SQL, and Flare for all TPC-H queries. For all systems, data loading time is excluded, i.e., only execution time is reported. In Spark and Flare, we use `persist` to ensure that the data is loaded from memory. At first glance, the performance of Flare and HyPer lie within the same range, and notably outperform Postgres and Spark in all queries. Similarly, Spark's performance is comparable to Postgres's in most of the queries. Unlike the other systems, Postgres does not compile queries at runtime, and relies on the Volcano model [23] for query evaluation, which incurs significant overhead. Hence, we can see that Spark's query compilation does not provide a significant advantage over a standard interpreted query engines on most queries.

At a closer look, Flare outperforms Spark SQL in aggregate queries Q1 and Q6 by 32× and 13× respectively. We observe that Spark is 200× slower than Flare in nested queries (e.g., Q2) After examining the execution plans of Q2, we found that Catalyst's plan does not detect all patterns that help with avoiding re-computations, e.g., a table which has been previously scanned or sorted. In join queries, e.g., Q5, Q10, Q14, etc., Flare is faster than Spark SQL by 19×-76×. Likewise, in join variants outer join Q13, semi-join Q21, and anti-join Q22, Flare is faster by 7×, 51× and 36× respectively.

The single-core performance gap between Spark SQL and Flare is attributed to the bottlenecks identified in Sections 2.3 and 2.4. First, overhead associated with low-level data access on the JVM. Second, Spark SQL's *distributed-first* strategy that employs costly distributed operators, e.g., sort-merge join and broadcast hash join, even when running on a single core. Third, internal bottlenecks in in-memory processing, the overhead of RDD operations, and communication through Spark's runtime system. By compiling entire queries, instead of isolated query stages, Flare effectively avoids these bottlenecks.

HyPer [35] is a state-of-the-art compiled relational query engine. A precursory look shows that Flare is faster than HyPer by 10%-60% in Q4-Q5,Q7, and Q14-Q16. Moreover, Flare is faster by 2× in Q3, Q11, and Q18. On the other hand, HyPer is faster than Flare by 20%-60% in Q9, Q10, Q12, and Q21. Moreover, HyPer is faster by 2×-4× in Q2, Q8, Q17, and Q20. This performance gap is, in part, attributed to (1) HyPer's use of specialized operators like GroupJoin [32], and (2) employing indexes on primary keys as seen in Q2, Q8, etc., whereas Flare (and Spark SQL) currently does not support indexes.

In summary, while both Flare and HyPer generate native code at runtime, subtle implementation differences in query evaluation and code generation can result in faster code. For instance, HyPer uses proper decimal precision numbers, whereas Flare follows Spark in using double precision floating point values which are native to the architecture. Furthermore, HyPer generates LLVM code, whereas Flare generates C code which is compiled with GCC.

Compilation Time We compared the compilation time for each TPC-H query on Spark and Flare (results omitted). For Spark, we measured the time to generate the physical plan, which includes Java code generation and compilation. We do not quantify JVM-internal JIT compilation, as this is hard to measure, and code may be recompiled multiple times. For Flare, we measured C code generation and compilation with GCC. Both sys-

tems spend a similar amount of time on code generation and compilation. Compilation time depends on the complexity of the query, but is less than 1s for all queries, i.e., well in line with interactive, exploratory, usage.

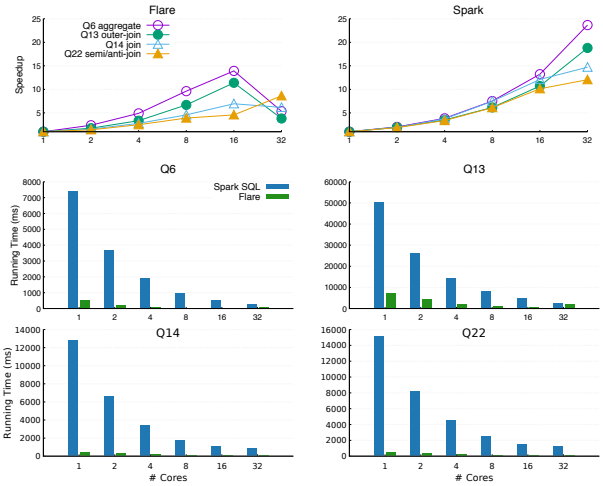


Figure 11: Scaling up Flare and Spark SQL in SF20, without NUMA optimizations: Spark has good nominal speedups (top), but Flare has better absolute running time in all configurations (bottom). For both systems, NUMA effects for 32 cores are clearly visible (Benchmark machine: 96 cores, 3 TB RAM across 4 CPU sockets, i.e., 24 cores, 750 GB each).

Parallel Scaling In this experiment, we compare the scalability of Spark SQL and Flare. The experiment focuses on the absolute performance and the Configuration that Outperforms a Single Thread (COST) metric proposed by McSherry et al. [30]. We pick four queries that represent aggregate and join variants.

Figure 11 presents speedup numbers for Q6, Q13, Q14, and Q22 when scaled up to 32 cores. At first glance, Spark appears to have good speedups in Q6 and Q13 whereas Flare's Q6 speedup drops for high core counts. However, examining the absolute running times, Flare is faster than Spark SQL by 14×. Furthermore, it takes Spark SQL estimated 16 cores in Q6 to match the performance of Flare's single core. In scaling up Q13, Flare is consistently faster by 6×-7× up to 16 cores. Similarly, Flare continues to outperform Spark by 12×-23× in Q14 and by 13×-22× in Q22.

What appears to be good scaling for Spark actually reveals that the runtime incurs significant overhead. In particular, we would expect Q6 to become memory-bound as we increase the level of parallelism. In Flare we can directly observe this effect as a sharp drop from 16 to 32 cores. Since our machine has 18 cores per socket, for 32 cores, we start accessing non-local memory (NUMA). The reason Spark scales better is because the internal overhead, which does not contribute anything to query evaluation, is trivially parallelizable and hides the memory bandwidth effects. In summary, Flare scales as expected for both of memory and CPU-bound workloads, and reflects the hardware characteristics of the workload, which means that query execution takes good advantage of the available resources – with the exception of multiple CPU sockets, a problem we address next.

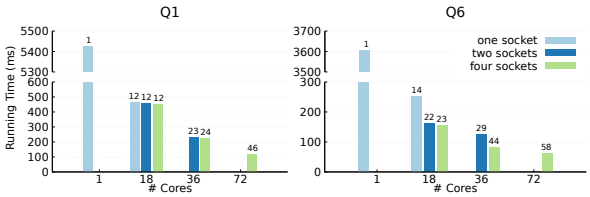


Figure 12: Scaling up Flare for SF100 with NUMA optimizations on different configurations: threads pinned to one, two, or four sockets. The speedups relative to a single thread are shown on top of the bars (Benchmark machine: 72 cores, 1 TB RAM across 4 CPU sockets, i.e., 18 cores, 250 GB each).

As a next step, we evaluate NUMA optimizations in Flare and show that these enable us to scale queries like Q6 to higher core numbers. In particular, we pin threads to individual cores and lay out memory such that most accesses are to the local memory region attached to each socket (Figure 12). Q6 performs better when the threads are dispatched on different sockets. This is due to the computation being bounded by the memory bandwidth. As such, when dividing the threads on multiple sockets, we multiply the available bandwidth proportionally. However, as Q1 is more computation bound, dispatching the threads on different sockets has little effect. For both Q1 and Q6, we see scaling up to the capacity of the machine (up to 72 cores). This is seen in a maximum speedup of 46× and 58× for Q1 and Q6, respectively.

Optimized Data Loading An often overlooked part of data processing is data loading. Flare contains an optimized implementation for both CSV files and the columnar Apache Parquet format.[2] We show loading times for each of the TPC-H tables in Table 1.

Full table read From the data in Table 1, we see that in both Spark and Flare, the Parquet file readers outperform the CSV file readers in most scenarios, despite this being a worst-case scenario for Parquet. Spark's CSV

[2]All Parquet files tested were uncompressed and encoded using PLAIN encoding.

Table	#Tuples	Postgres CSV	HyPer CSV	Spark CSV	Spark Parquet	Flare CSV	Flare Parquet
CUSTOMER	1500000	7067	1102	11664	9730	329	266
LINEITEM	59986052	377765	49408	471207	257898	11167	10668
NATION	25	1	8	106	110	< 1	< 1
ORDERS	15000000	60214	33195	85985	54124	2028	1786
PART	2000000	8807	1393	11154	7601	351	340
PARTSUPP	8000000	37408	5265	28748	17731	1164	1010
REGION	5	1	8	102	90	< 1	< 1
SUPPLIER	100000	478	66	616	522	28	16

Table 1: Loading time in ms for TPC-H SF10 in Postgres, HyPer, Flare, and SparkSQL.

reader was faster in only one case: reading `nation`, a table with only 25 rows. In all other cases, Spark's Parquet reader was 1.33×-1.81× faster. However, Flare's highly optimized CSV reader operates at a closer level of performance to the Parquet reader, with all tables except `supplier` having a benefit of less than a 1.25× speedup by using Parquet.

Performing queries Figure 13 shows speedups gained from executing queries without preloading data for both systems. Whereas reading an entire table gives Spark and Flare marginal speedups, reading just the required data gives speedups in the range of 2×-144× (Q16 remained the same) for Spark and 60%-14× for Flare. Across systems, Flare's Parquet reader demonstrated between a 2.5×-617× speedup over Spark's, and between 34×-101× over Spark's CSV reader. While the speedup over Spark lessens slightly in higher scale factors, we found that Flare's Parquet reader consistently performed on average at least one order of magnitude faster across each query, regardless of scale factor.

In nearly every case, reading from a Parquet file in Flare is approximately 2×-4× slower than in-memory processing. However, reading from a Parquet file in Spark is rarely significantly slower than in-memory processing. These results show that while reading from Parquet certainly provides performance gains for Spark when compared to reading from CSV, the overall performance bottleneck of Spark does not lie in the cost of reading from SSD compared to in-memory processing.

TensorFlow We evaluate the performance of Flare and TensorFlow integration with Spark. We run the query shown in Figure 9, which embeds a UDF that performs classification via machine learning over the data (based on a pre-trained model). As shown in Figure 14, using Flare in conjunction with TensorFlow provides speedups of over 1,000,000× when compared to PySpark, and 60× when Spark calls the TensorFlow UDF through JNI. Thus, while we can see that interfacing with an object file gives an important speed-up to Spark, the data loading ultimately becomes the bottleneck for the system. Flare, however, can optimize the data layout to reduce the amount of data copied to the bare minimum, and eliminate essentially all of the inefficiencies on the boundary between Spark and TensorFlow.

6 Related Work

Cluster Computing Frameworks Such frameworks typically implement a combination of parallel, distributed, relational, procedural, and MapReduce computations. The MapReduce model [18] realized in Hadoop [6] performs big data analysis on shared-nothing, potentially unreliable, machines. Twister [20] and Haloop [14] support iterative MapReduce workloads by avoiding reading unnecessary data and keeping invariant data between iterations. Likewise, Spark [55, 56] tackles the issue of data reuse among MapReduce jobs or applications by explicitly persisting intermediate results in memory. Along the same lines, the need for an expressive programming model to perform analytics on structured and semistructured data motivated Hive [52], Dremel [31], Impala [26], Shark [53] and Spark SQL [8] and many others. SnappyData [41] integrates Spark with a transactional main-memory database to realize a unified engine that supports streaming, analytics and transactions. Asterix [10], Stratosphere / Apache Flink [5], and Tupleware [17] are other systems that improve over Spark in various dimensions, including UDFs and performance, and which inspired the design of Flare. While these systems are impressive, Flare sets itself apart by accelerating actual Spark workloads instead of proposing a competing system, and by demonstrating relational performance on par with HyPer [35] on the full set of TPC-H queries. Moreover, in contrast to systems like Tupleware that mainly integrate UDFs on the LLVM level, Flare uses higher-level knowledge about specific external systems, such as TensorFlow. Similar to Tupleware, Flare's main target are small clusters of powerful machines where faults are statistically improbable.

Query Compilation Recently, code generation for SQL queries has regained momentum. Historic efforts go back all the way to System R [9]. Query compilation can be realized using code templates e.g., Spade[22] or HIQUE [27], general purpose compilers, e.g., HyPer [35] and Hekaton [19], or DSL compiler frameworks, e.g., Legobase [25], DryadLINQ [54], DBLAB [45], and LB2 [49].

Embedded DSL Frameworks and Intermediate Languages These address the compromise between productivity and performance in writing programs that can run under diverse programming models. Voodoo [39] addresses compiling portable query plans that can run

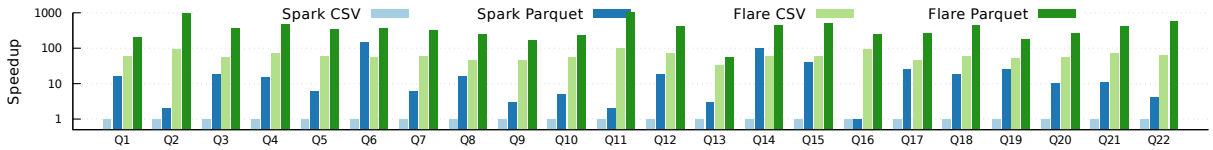


Figure 13: Speedup for TPC-H SF1 when streaming data from SSD on a single thread.

#Data Points	Spark	Spark + JNI	Flare
200	11909	990	0.064
2000	522471	3178	0.503

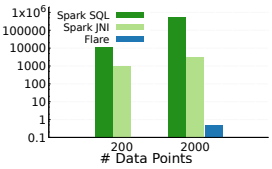


Figure 14: Running time (ms) of query in Figure 9 using TensorFlow in Spark and Flare.

on CPUs and GPUs. Voodoo's intermediate algebra is expressive and captures hardware optimizations, e.g., multicores, SIMD, etc. Furthermore, Voodoo is used as an alternative back-end for MonetDB [12]. Delite [46], a general purpose compiler framework, implements high-performance DSLs (e.g., SQL, Machine Learning, graphs and matrices), provides parallel patterns and generates code for heterogeneous targets. The Distributed Multiloop Language (DMLL) [13] provides rich collections and parallel patterns and supports big-memory NUMA machines. Weld [38] is another recent system that aims to provide a common runtime for diverse libraries e.g., SQL and machine learning. Steno [33] performs optimizations similar to DMLL to compile LINQ queries. Furthermore, Steno uses DryadLINQ [54] runtime for distributed execution. Nagel et. al. [34] generates efficient code for LINQ queries. Weld is similar to DMLL in supporting nested parallel structures.

Performance evaluation In data analytics frameworks, performance evaluation aims to identify bottlenecks and study the parameters that impact performance the most, e.g., workload, scale-up/scale-out resources, probability of faults, etc. A recent study [36] on a single Spark cluster revealed that CPU, not I/O, is the source of bottlenecks. McSherry et al. [30] proposed the COST (Configuration that Outperforms a Single Thread) metric, and showed that in many cases, single-threaded programs can outperform big data processing frameworks running on large clusters. TPC-H [51] is a decision support benchmark that consists of 22 analytical queries that address several "choke points," e.g., aggregates, large joins, arithmetic computations, etc. [11].

7 Conclusions

Modern data analytics need to combine multiple programming models and make efficient use of modern hardware with large memory, many cores, and NUMA capabilities. We introduce Flare: a new backend for Spark that brings relational performance on par with the best SQL engines, and also enables highly optimized heterogeneous workloads with external ML systems. Most importantly, all of this comes without giving up the expressiveness of Spark's high-level APIs. We believe that multi-stage APIs, in the spirit of DataFrames, and compiler systems like Flare, will play an increasingly important role in the future to satisfy the increasing demand for flexible and unified analytics with high efficiency.

Acknowledgments

This work was supported in part by NSF awards 1553471 and 1564207, DOE award DE-SC0018050, and a Google Faculty Research Award.

References

[1] ONNX. https://github.com/onnx/onnx.

[2] OpenMP. http://openmp.org/.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems, 2015.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[5] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[6] Apache. Hadoop. http://hadoop.apache.org/.

[7] Apache. Parquet. https://parquet.apache.org/.

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

[9] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, et al. System R: relational approach to database management. *TODS*, 1(2):97–137, 1976.

[10] A. Behm, V. R. Borkar, M. J. Carey, R. Grover, C. Li, N. Onose, R. Vernica, A. Deutsch, Y. Papakonstantinou, and V. J. Tsotras. Asterix: towards a scalable, semistructured data platform for evolving-world models. *Distributed and Parallel Databases*, 29(3):185–216, 2011.

[11] P. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 61–76. Springer, 2013.

[12] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005.

[13] K. J. Brown, H. Lee, T. Rompf, A. K. Sujeeth, C. De Sa, C. Aberger, and K. Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. CGO 2016, pages 194–205. ACM, 2016.

[14] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: efficient iterative data processing on large clusters. *PVLDB*, 3(1-2):285–296, 2010.

[15] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. GPCE, pages 57–76, 2003.

[16] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[17] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Çetintemel, and S. Zdonik. An architecture for compiling UDF-centric workflows. *PVLDB*, 8(12):1466–1477, 2015.

[18] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[19] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL server's memory-optimized oltp engine. In *SIGMOD*, pages 1243–1254. ACM, 2013.

[20] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative MapReduce. In *ACM HPCD*, pages 810–818. ACM, 2010.

[21] Y. Futamura. Partial evaluation of computation process — an approach to a compiler-compiler. *Transactions of the Institute of Electronics and Communication Engineers of Japan*, 54-C(8):721–728, 1971.

[22] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the System S declarative stream processing engine. In *SIGMOD*, pages 1123–1134. ACM, 2008.

[23] G. Graefe. Volcano-an extensible and parallel query evaluation system. *TKDE*, 6(1):120–135, 1994.

[24] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[25] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[26] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. a. Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR*, 2015.

[27] K. Krikellas, S. D. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624. IEEE, 2010.

[28] J. Laskowski. Mastering Apache Spark 2. https://www.gitbook.com/book/jaceklaskowski/mastering-apache-spark/details, 2016.

[29] F. McSherry. Scalability! but at what COST. http://www.frankmcsherry.org/graph/scalability/cost/2015/01/15/COST.html, 2015.

[30] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *HotOS*, 2015.

[31] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, 2010.

[32] G. Moerkotte and T. Neumann. Accelerating queries with group-by and join by groupjoin. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[33] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In *ACM SIGPLAN Notices*, volume 46, pages 121–131, 2011.

[34] F. Nagel, G. Bierman, and S. D. Viglas. Code generation for efficient query processing in managed runtimes. *VLDB*, 7(12):1095–1106, 2014.

[35] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[36] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B. Chun. Making sense of performance in data analytics frameworks. In *NSDI*, pages 293–307, 2015.

[37] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[38] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab. Weld: A common runtime for high performance data analytics. In *CIDR*, 2017.

[39] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo - a vector algebra for portable database performance on modern hardware. *VLDB*, 9(14):1707–1718, 2016.

[40] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. Galindo-Legaria, and C. Cunningham. Froid: Optimization of imperative programs in a relational database. *Proceedings of the VLDB Endowment*, 11(4), 2017.

[41] J. Ramnarayan, B. Mozafari, S. Wale, S. Menon, N. Kumar, H. Bhanawat, S. Chakraborty, Y. Mahajan, R. Mishra, and K. Bachhav. Snappydata: A hybrid transactional analytical store built on Spark. In *SIGMOD*, pages 2153–2156, 2016.

[42] T. Rompf and N. Amin. Functional Pearl: A SQL to C compiler in 500 lines of code. In *ICFP*, 2015.

[43] T. Rompf and M. Odersky. Lightweight Modular Staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[44] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas. Nobody ever got fired for using Hadoop on a cluster. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, HotCDP '12, pages 2:1–2:5, New York, NY, USA, 2012. ACM.

[45] A. Shaikhha, Y. Klonatos, L. Parreaux, L. Brown, M. Dashti, and C. Koch. How to architect a query compiler. In *SIGMOD*, pages 1907–1922. ACM, 2016.

[46] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *TECS*, 13(4s):134, 2014.

[47] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[48] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[49] R. Y. Tahboub, G. M. Essertel, and T. Rompf. How to architect a query compiler, revisited. SIGMOD '18, pages 307–322. ACM, 2018.

[50] T. X. Team. XLA – TensorFlow Compiled. Post on the Google Developers Blog, 2017. http://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html.

[51] The Transaction Processing Council. TPC-H Version 2.15.0.

[52] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.

[53] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *ACM SIGMOD*, pages 13–24, 2013.

[54] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. *OSDI*, 8, 2008.

[55] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *USENIX*, HotCloud'10, 2010.

[56] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016.