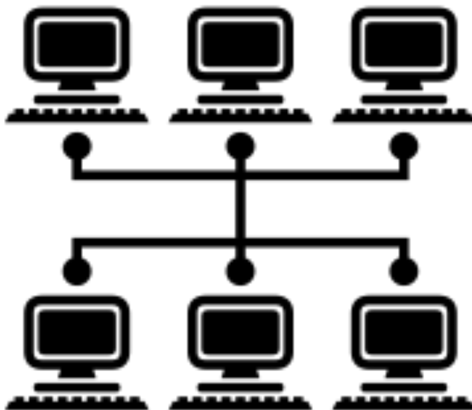
Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data

Gregory 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

APACHE
Spark
TM



How Fast Is Spark?

Search Blog

COMPANY BLOG

- Announcements
- Customers
- Events
- Partners
- Product
- Security

ENGINEERING BLOG

- Apache Spark
- Ecosystem
- Machine Learning
- Platform
- Streaming

SEE ALL

SUBSCRIBE

Blog

Newsletter

FOLLOW

Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop

Deep dive into the new Tungsten execution engine

by Sameer Agarwal, Davies Liu and Reynold Xin
Posted in **ENGINEERING BLOG** | May 23, 2016

Try this notebook in Databricks

When our team at Databricks planned our contributions to the upcoming Apache Spark 2.0 release, we set out with an ambitious goal by asking ourselves: **Apache Spark is already pretty fast, but can we make it 10x faster**?

This question led us to fundamentally rethink the way we built Spark's physical execution layer. When you look into a modern data engine (e.g. Spark or other MPP databases), a majority of the CPU cycles are spent in useless work, such as making virtual function calls or reading or writing intermediate data to CPU cache or memory. Optimizing performance by reducing the amount of CPU cycles wasted in this useless work has been a long-time focus of modern compilers.

Apache Spark 2.0 will ship with the second generation Tungsten engine. Built upon ideas from modern compilers and MPP databases and applied to data processing queries, Tungsten emits (SPARK-12795) optimized bytecode at runtime that collapses the entire query into a single function, eliminating virtual function calls and leveraging CPU registers for intermediate data. As a result of this streamlined strategy, called "whole-stage code generation," we significantly improve CPU efficiency and gain performance.

SHARE POST

REFERENCES

- Efficiently Compiling Efficient Query Plans for Modern Hardware (VLDB 2011)
- The Morning Paper: Efficiently Compiling Efficient Query Plans for Modern Architecture
- Volcano-An Extensible and Parallel Query Evaluation System (TKDE 1990)

Motivation

96 cores and **3** TB of RAM (on 4 sockets)

Let’s dive into Spark

Efficiently Compiling Efficient Query Plans for Modern Hardware

ABSTRACT

As main memory grows, query perfo
determined by the raw CPU costs
The classical iterator style query pr
simple and flexible, but shows poor
CPUs due to lack of locality and
predictions. Several techniques like
or vectorized tuple processing hav

Spark SQL: Relational Data Processing in Spark

Michael Armbrust[†], Reynold S. Xin[†], Cheng Lian[†], Yin Huai[†], Davies Liu[†], Joseph K. Bradley[†], Xiangrui Meng[†], Tomer Kaftan[‡], Michael J. Franklin[†‡], Ali Ghodsi[†], Matei Zaharia[†*]

[†]Databricks Inc. [*]MIT CSAIL [‡]AMPLab, UC Berkeley

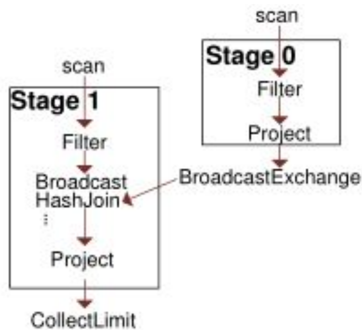
ABSTRACT

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark's functional programming API. Built on our experience with Shark, Spark SQL lets Spark programmers leverage the benefits of relational processing (*e.g.,* declarative queries and optimized storage), and lets SQL users call complex analytics libraries in Spark (*e.g.,* machine learning). Compared to previous systems, Spark SQL makes two main additions. First, it offers much tighter integration between relational and procedu-

While the popularity of relational systems shows that users often prefer writing declarative queries, the relational approach is insufficient for many big data applications. First, users want to perform ETL to and from various data sources that might be semi- or unstructured, requiring custom code. Second, users want to perform advanced analytics, such as machine learning and graph processing, that are challenging to express in relational systems. In practice, we have observed that most data pipelines would ideally be expressed with a combination of both relational queries and

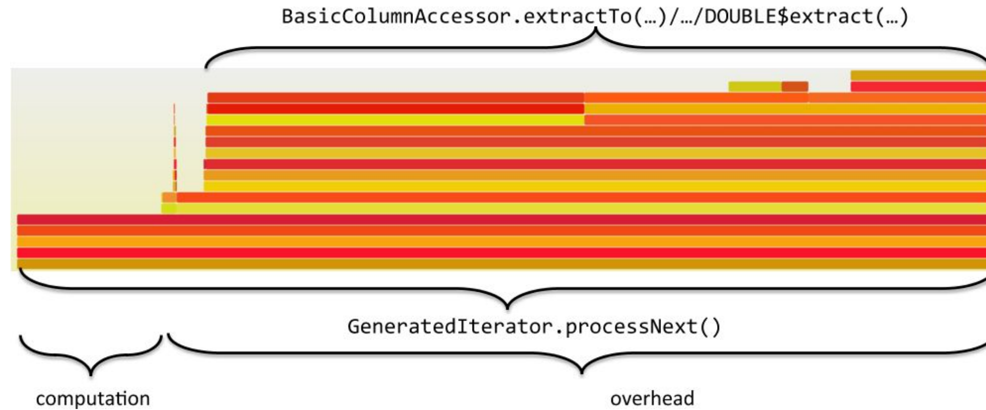
Bottlenecks

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

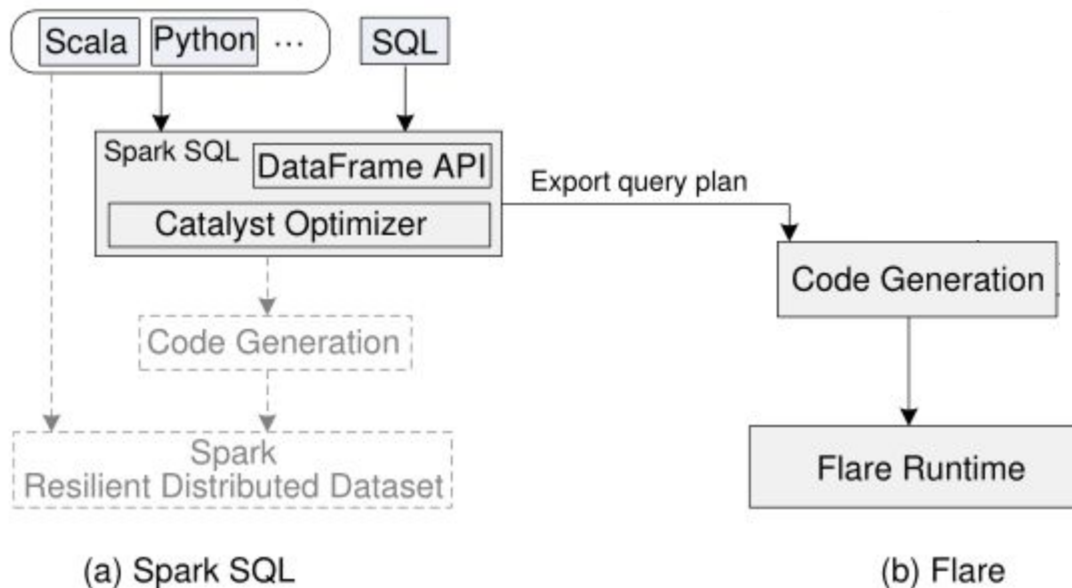



```
override def doConsume(ctx: CodegenContext, input: Seq[ExprCode],
    row: ExprCode): String = {
  val (broadcastRelation, relationTerm) = prepareBroadcast(ctx)
  val (keyEv, anyNull) = genStreamSideJoinKey(ctx, input)
  val (matched, checkCondition, buildVars) = getJoinCondition(ctx, input)
  val 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
}
```

Bottlenecks



Flare: a New Back-End for Spark



Flare design

How to Architect a Query Compiler, Revisited

Ruby Y. Tahboub, Grégory M. Essertel, Tiark Rompf
Purdue University, West Lafayette, Indiana
{rtahboub,gesserte,tiark}@purdue.edu

ABSTRACT

To leverage modern hardware platforms to their fullest, more and more database systems embrace compilation of query plans to native code. In the research community, there is an ongoing debate about the best way to architect such query compilers. This is perceived to be a difficult task, requiring techniques fundamentally different from traditional interpreted query execution.

We aim to contribute to this discussion by drawing attention to an old but underappreciated idea known as *Futamura projections*, which fundamentally link interpreters and compilers. Guided by this idea, we demonstrate that efficient query compilation can actually be very simple, using techniques that are no more difficult than writing a query interpreter in a high-level language. Moreover

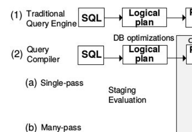


Figure 1: Illustration of (1)
pilers (a) single-pass compi

rewrites this plan into a more
and emits a physical plan rea
usually an interpreter that ex

Functional Pearl: A SQL to C Compiler in 500 Lines of Code

Tiark Rompf * Nada Amin‡
*Purdue University, USA: {first}@purdue.edu
‡EPFL, Switzerland: {first.last}@epfl.ch

Abstract

We present the design and implementation of a SQL query processor that outperforms existing database systems and is written in just about 500 lines of Scala code – a convincing case study that high-level functional programming can handily beat C for systems-level programming where the last drop of performance matters.

'a'. As a first step to perform further data analysis, we load this file into a database system, for example MySQL:

```
mysqlimport --local mydb 1gram_a.csv
```

When we run this command we can safely take a coffee break, as the import will take a good five minutes on a decently modern

Lightweight Modular Staging (LMS)

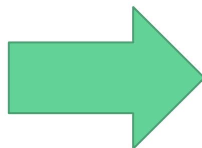
```
def power(x: Int, n: Int): Int = {
  if (n == 0)
    1
  else
    x * power(x, n - 1)
}
```

Stage

```
def power(x: Rep[Int], n: Int): Rep[Int] = {
  if (n == 0)
    1
  else
    x * power(x, n - 1)
}

val x: Rep[Int] = ...
val res = power(x, 4)
```

Run



```
val x: Int = ...
val x1 = x * x
val x2 = x * x1
val x3 = x * x2
val res = x3
```

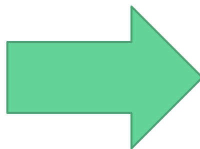
Flare implementation

```
type Pred = Record => Rep[Boolean]

abstract class Op {
  def exec(callback: Record => Unit): Unit
}

class Select(op: Op)(pred: Pred) extends Op {
  def exec(cb: Record => Unit) = {
    op.exec { tuple =>
      if (pred(tuple)) cb(tuple)
    }
  }
}
```

```
select
  l_returnflag
from
  lineitem
where
   l_quantity <= 1
```



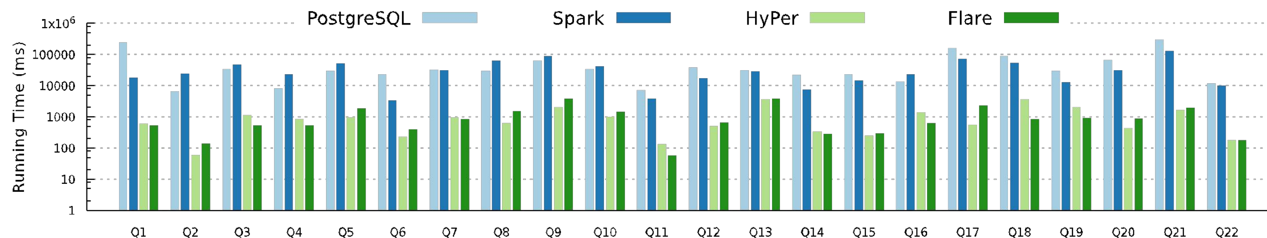
```
double* x14 = ... // data loading l_quantity
char* x22 = ...    // l_returnflag
printf("%s\n","begin scan lineitem");
for (x744 = 0; x744 < x4; x744++) {
    long x760 = x744;
    double* x769 = x14;
    double x770 = x769[x760];
    char* x777 = x22;
    char x778 = x777[x760];
    bool x804 = x770 <= 1.0;
    if (x804) {
        printf("%c|\n",x778);
    }
}
```

Results

Single-Core Running Time: TPCH

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Postgres	241404	6649	33721	7936	30043	23358	32501	29759	64224	33145	7093
Spark	18219	23741	47816	22630	51731	3383	31770	63823	88861	42216	3857
Hyper	603	59	1126	842	941	232	943	616	1984	967	131
Flare	529	139	536	520	747	365	828	1533	3131	1795	56

	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Postgres	37880	31242	22058	23133	13232	155449	90949	29452	65541	299178	11703
Spark	17233	28489	7403	14542	23371	70944	53932	13085	31226	173778	10030
Hyper	501	3625	330	253	1399	563	3703	1980	434	1626	180
Flare	654	3715	260	303	620	2338	825	908	870	1963	177

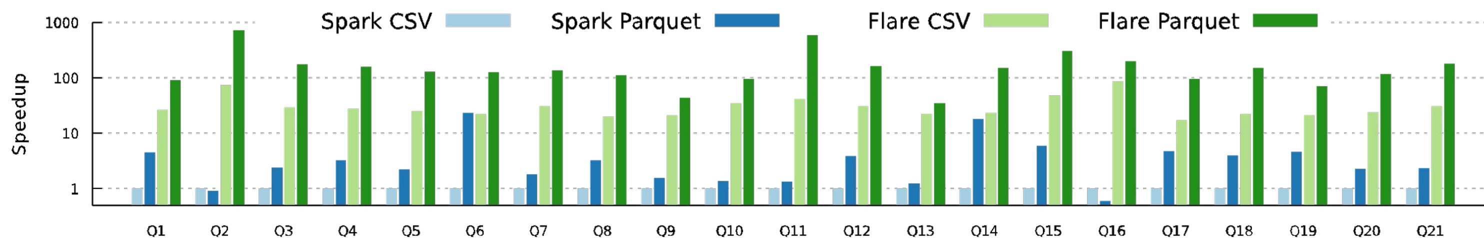


Absolute running time in milliseconds (ms) for Postgres, Spark, HyPer and Flare in SF10

Apache Parquet Format

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
Spark CSV	16762	12244	21730	19836	19316	12278	24484	17726	30050	29533	5224
Spark Parquet	3728	13520	9099	6083	8706	535	13555	5512	19413	21822	3926
Flare CSV	641	168	757	698	758	568	788	875	1417	854	128
Flare Parquet	187	17	125	127	151	99	183	160	698	309	9

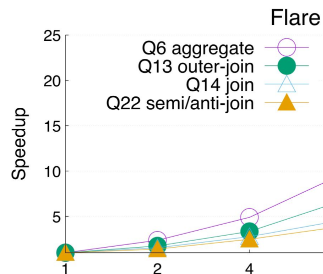
	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Spark CSV	21688	8554	12962	26721	12941	24690	27012	12409	19369	57330	7050
Spark Parquet	5570	7034	719	4506	21834	5176	6757	2681	8562	25089	5295
Flare CSV	701	388	573	551	150	1426	1229	605	792	1868	178
Flare Parquet	133	246	86	88	66	264	181	178	165	324	22

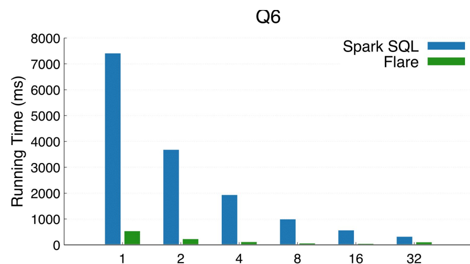


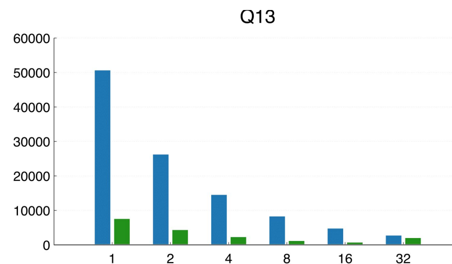
What about parallelism?

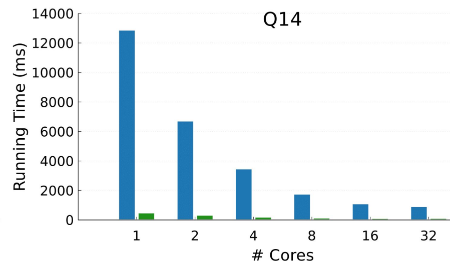
Parallel Scaling Experiment

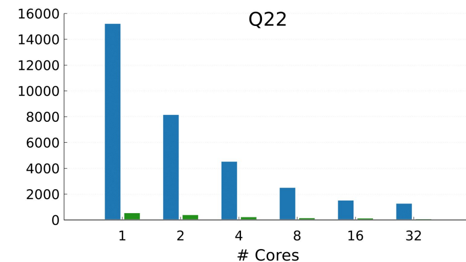
Scaling-up Flare and Spark SQL in SF20





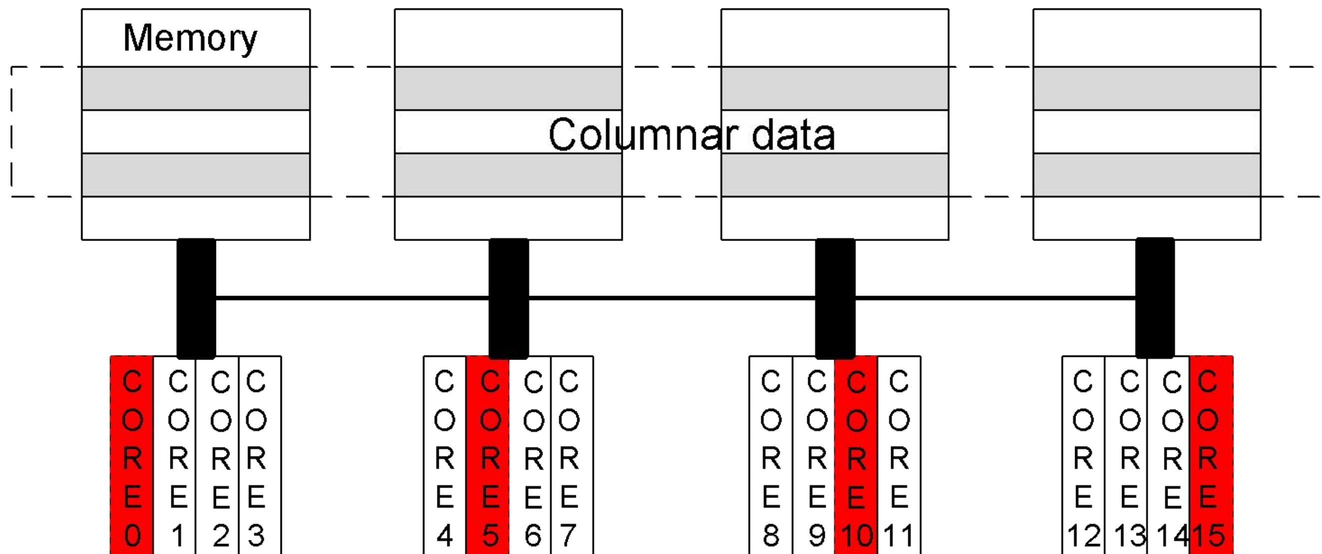




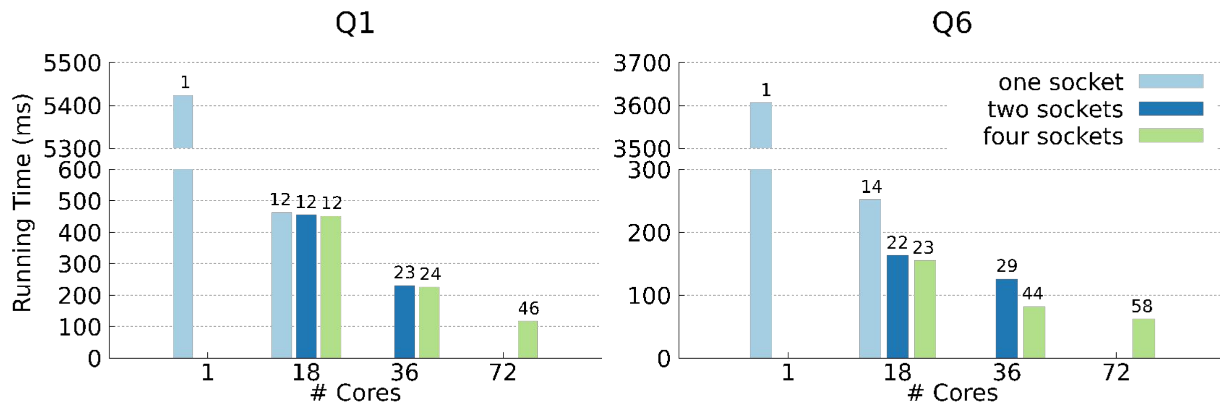


Hardware: Single NUMA machine with 4 sockets, 24 Xeon Platinum 8168 @ 2.70GHz cores per socket, and 256GB RAM per socket (1 TB total).

NUMA Optimization



NUMA Optimization

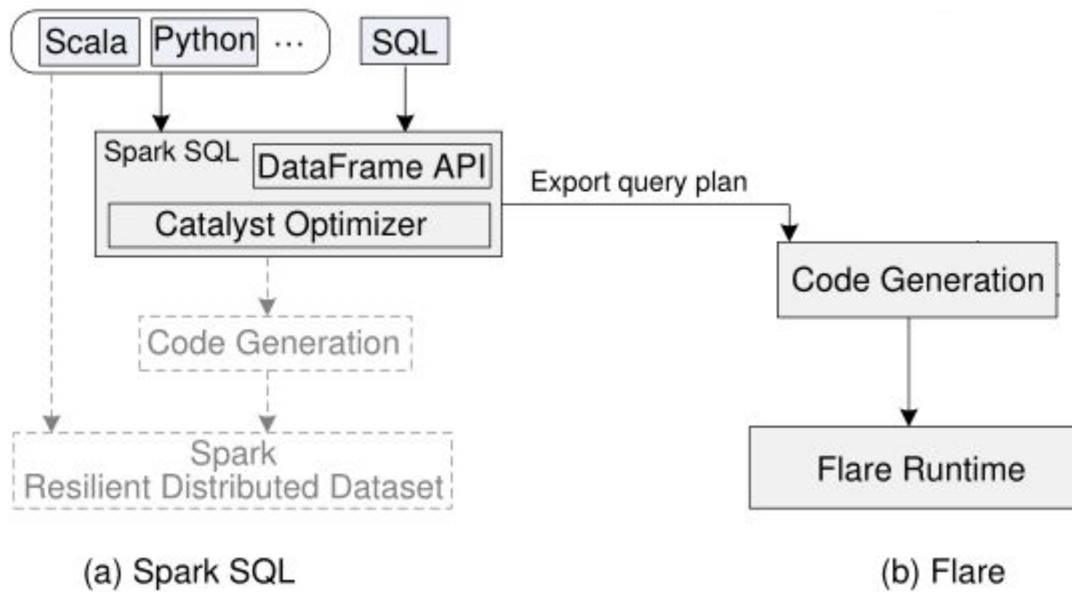


Scaling-up Flare for SF100 with NUMA optimizations on different configurations: threads pinned to one, two and four sockets

Hardware: Single NUMA machine with 4 sockets, 18 Xeon E5-4657L cores per socket, and 256GB RAM per socket (1 TB total).

Heterogeneous Workloads: UDFs and ML Kernels

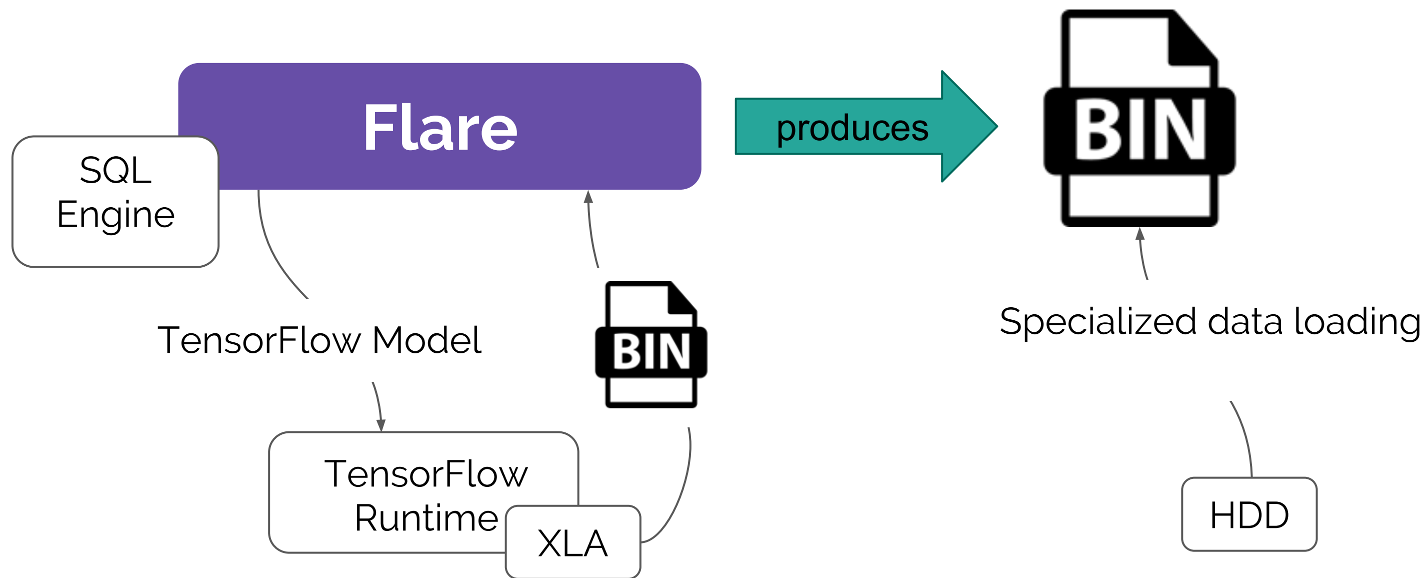
Flare



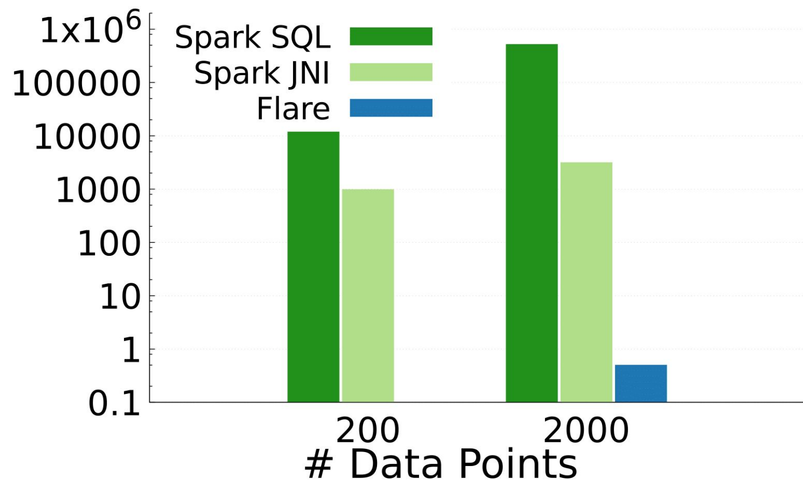
(a) Spark SQL

(b) Flare

TensorFlare architecture



Results




```
# 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
spark.udf.register("classifier", classifier)
# Use classifier in PySpark:
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")
q.show()
```

flaredata.github.io

- Identify key impediments to performance for medium-sized workloads running on Spark
- Flare optimizes data loading and generates parallel code NUMA aware
- Flare reduces the gap between Spark SQL and best-of-breed relational query engines

Thank you!