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

Gregory Essertel\textsuperscript{1}, Ruby Y. Tahboub\textsuperscript{1}, James M. Decker\textsuperscript{1}, Kevin J. Brown\textsuperscript{2}, Kunle Olukotun\textsuperscript{2}, Tiark Rompf\textsuperscript{1}

\textsuperscript{1}Purdue University, \textsuperscript{2}Stanford University
How Fast Is Spark?
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.
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 performance is determined by the raw CPU cost of operations. The classical iterator style query processing is simple and flexible, but shows poor performance on modern CPUs due to lack of locality and poor cache predictions. Several techniques like query blocking or vectorized tuple processing help to mitigate performance issues.

Spark SQL: Relational Data Processing in Spark

Michael Armbrust*, Reynold S. Xin†, Cheng Lian†, Yin Hui†, Davies Liu†, Joseph K. Bradley†, Xiangrui Meng†, Tomer Kalvan†, 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 procedure-oriented approaches. Second, it employs a number of novel techniques to achieve high performance.

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

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

"""
|// 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

BasicColumnAccessor.extractTo(...) / DOUBLE$extract(...)

GeneratedIterator.processNext()

computation

overhead
Flare: a New Back-End for Spark

![Diagram showing the Flare architecture]
Flare design

How to Architect a Query Compiler, Revisited
Ruby Y. Tahboub, Grégoire M. Essertel, Tiark Rompf
Purdue University, West Lafayette, Indiana
{tahboub, essertel, 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 more intricate in a high-level language. Moreover...

Figure 1: Illustration of (1) pipers (a) single-pass compiler
rewrites this plan into a more
and emits a physical plan res-
ually an interpreter that an

Functional Pearl: A SQL to C Compiler in 500 Lines of Code
Tiark Rompf 1
Purdue University, USA; (first@purdue.edu)
Nada Amin 2
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 handle even 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 mysql igram_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)

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

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)
```

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

```scala
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
```

```plaintext
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

<table>
<thead>
<tr>
<th>Q1</th>
<th>Q2</th>
<th>Q3</th>
<th>Q4</th>
<th>Q5</th>
<th>Q6</th>
<th>Q7</th>
<th>Q8</th>
<th>Q9</th>
<th>Q10</th>
<th>Q11</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postgres</td>
<td>241404</td>
<td>6649</td>
<td>33721</td>
<td>7936</td>
<td>30043</td>
<td>23358</td>
<td>32501</td>
<td>29759</td>
<td>64224</td>
<td>33145</td>
</tr>
<tr>
<td>Spark</td>
<td>18219</td>
<td>23741</td>
<td>47816</td>
<td>22630</td>
<td>51731</td>
<td>3383</td>
<td>31770</td>
<td>63823</td>
<td>88861</td>
<td>42216</td>
</tr>
<tr>
<td>HyPer</td>
<td>603</td>
<td>59</td>
<td>1126</td>
<td>842</td>
<td>941</td>
<td>232</td>
<td>943</td>
<td>616</td>
<td>1984</td>
<td>967</td>
</tr>
<tr>
<td>Flare</td>
<td>529</td>
<td>139</td>
<td>536</td>
<td>520</td>
<td>747</td>
<td>365</td>
<td>828</td>
<td>1533</td>
<td>3131</td>
<td>1795</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q12</th>
<th>Q13</th>
<th>Q14</th>
<th>Q15</th>
<th>Q16</th>
<th>Q17</th>
<th>Q18</th>
<th>Q19</th>
<th>Q20</th>
<th>Q21</th>
<th>Q22</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postgres</td>
<td>37880</td>
<td>31242</td>
<td>22058</td>
<td>23133</td>
<td>13232</td>
<td>155449</td>
<td>90949</td>
<td>29452</td>
<td>65541</td>
<td>299178</td>
</tr>
<tr>
<td>Spark</td>
<td>17233</td>
<td>28489</td>
<td>7403</td>
<td>14542</td>
<td>23371</td>
<td>70944</td>
<td>53932</td>
<td>13085</td>
<td>31226</td>
<td>173778</td>
</tr>
<tr>
<td>HyPer</td>
<td>501</td>
<td>3625</td>
<td>330</td>
<td>253</td>
<td>1399</td>
<td>563</td>
<td>3703</td>
<td>1980</td>
<td>434</td>
<td>1626</td>
</tr>
<tr>
<td>Flare</td>
<td>654</td>
<td>3715</td>
<td>260</td>
<td>303</td>
<td>620</td>
<td>2338</td>
<td>825</td>
<td>908</td>
<td>870</td>
<td>1963</td>
</tr>
</tbody>
</table>

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

### Table 1: Comparison of Spark and Flare CSV/Parquet for Q1 to Q10

<table>
<thead>
<tr>
<th></th>
<th>Spark CSV</th>
<th>Spark Parquet</th>
<th>Flare CSV</th>
<th>Flare Parquet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>16762</td>
<td>12244</td>
<td>641</td>
<td>187</td>
</tr>
<tr>
<td>Q2</td>
<td>12244</td>
<td>21730</td>
<td>168</td>
<td>17</td>
</tr>
<tr>
<td>Q3</td>
<td>21730</td>
<td>19836</td>
<td>757</td>
<td>125</td>
</tr>
<tr>
<td>Q4</td>
<td>19836</td>
<td>19316</td>
<td>698</td>
<td>127</td>
</tr>
<tr>
<td>Q5</td>
<td>19316</td>
<td>12278</td>
<td>758</td>
<td>151</td>
</tr>
<tr>
<td>Q6</td>
<td>12278</td>
<td>24484</td>
<td>568</td>
<td>99</td>
</tr>
<tr>
<td>Q7</td>
<td>24484</td>
<td>17726</td>
<td>788</td>
<td>183</td>
</tr>
<tr>
<td>Q8</td>
<td>17726</td>
<td>30050</td>
<td>875</td>
<td>160</td>
</tr>
<tr>
<td>Q9</td>
<td>30050</td>
<td>30050</td>
<td>1417</td>
<td>698</td>
</tr>
<tr>
<td>Q10</td>
<td>30050</td>
<td>29533</td>
<td>854</td>
<td>309</td>
</tr>
</tbody>
</table>

### Table 2: Comparison of Spark and Flare CSV/Parquet for Q12 to Q22

<table>
<thead>
<tr>
<th></th>
<th>Spark CSV</th>
<th>Spark Parquet</th>
<th>Flare CSV</th>
<th>Flare Parquet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q12</td>
<td>21688</td>
<td>8554</td>
<td>701</td>
<td>133</td>
</tr>
<tr>
<td>Q13</td>
<td>8554</td>
<td>12962</td>
<td>388</td>
<td>246</td>
</tr>
<tr>
<td>Q14</td>
<td>12962</td>
<td>26721</td>
<td>573</td>
<td>86</td>
</tr>
<tr>
<td>Q15</td>
<td>26721</td>
<td>12941</td>
<td>551</td>
<td>88</td>
</tr>
<tr>
<td>Q16</td>
<td>12941</td>
<td>24690</td>
<td>150</td>
<td>66</td>
</tr>
<tr>
<td>Q17</td>
<td>24690</td>
<td>27012</td>
<td>1426</td>
<td>264</td>
</tr>
<tr>
<td>Q18</td>
<td>27012</td>
<td>12409</td>
<td>1229</td>
<td>181</td>
</tr>
<tr>
<td>Q19</td>
<td>12409</td>
<td>19369</td>
<td>605</td>
<td>178</td>
</tr>
<tr>
<td>Q20</td>
<td>19369</td>
<td>57330</td>
<td>792</td>
<td>165</td>
</tr>
<tr>
<td>Q21</td>
<td>57330</td>
<td>7050</td>
<td>1868</td>
<td>324</td>
</tr>
<tr>
<td>Q22</td>
<td>7050</td>
<td>7050</td>
<td>178</td>
<td>22</td>
</tr>
</tbody>
</table>

### Graph 1: Speedup Comparison

- **Spark CSV**
- **Spark Parquet**
- **Flare CSV**
- **Flare Parquet**
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
import tensorflow as tf

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()
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