Project Tungsten Phase II

Joining a Billion Rows per Second on a Laptop

Sameer Agarwal

Spark Meetup | SAP | June 30th 2016

databricks

About Me

- Software Engineer at Databricks (Spark Core/SQL)
- PhD in Databases (UC Berkeley)
- Research on BlinkDB (Approximate Queries in Spark)

Hardware Trends

Storage

Network

CPU

Hardware Trends

	2010
Storage	50+MB/s (HDD)
Network	1Gbps
CPU	~3GHz

Hardware Trends

	2010	2016
Storage	50+MB/s (HDD)	500+MB/s (SSD)
Network	1Gbps	10Gbps
CPU	~3GHz	~3GHz

Hardware Trends

	2010	2016	
Storage	50+MB/s (HDD)	500+MB/s (SSD)	10X
Network	1Gbps	10Gbps	10X
CPU	~3GHz	~3GHz	🙁

On the flip side

Spark IO has been optimized

- Reduce IO by pruning input data that is not needed
- New shuffle and network implementations (2014 sort record)

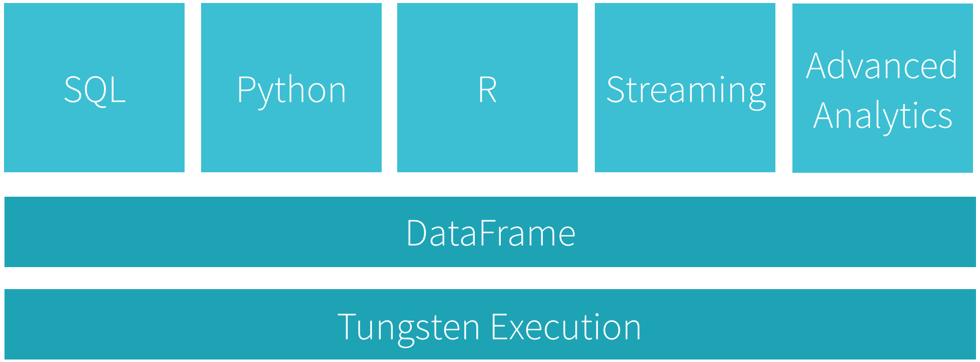
Data formats have improved

- E.g. Parquet is a “dense” columnar format

CPU increasingly the bottleneck; trend expected to continue

Goals of Project Tungsten

Substantially improve the **memory and CPU** efficiency of Spark backend execution and push performance closer to the limits of modern hardware.



Note the focus on "execution" not "optimizer": very easy to pick broadcast join that is 1000X faster than Cartesian join, but hard to optimize broadcast join to be an order of magnitude faster.

Phase 1

Foundation

Memory Management

Code Generation

Cache-aware Algorithms

Phase 2

Order-of-magnitude Faster

Whole-stage Codegen

Vectorization

Phase 1
Laying The Foundation

Summary

Perform manual memory management instead of relying on Java objects

- Reduce memory footprint
- Eliminate garbage collection overheads
- Use java.unsafe and off heap memory

Code generation for expression evaluation

- Reduce virtual function calls and interpretation overhead

Cache conscious sorting

- Reduce bad memory access patterns

Phase 2
Order-of-magnitude Faster

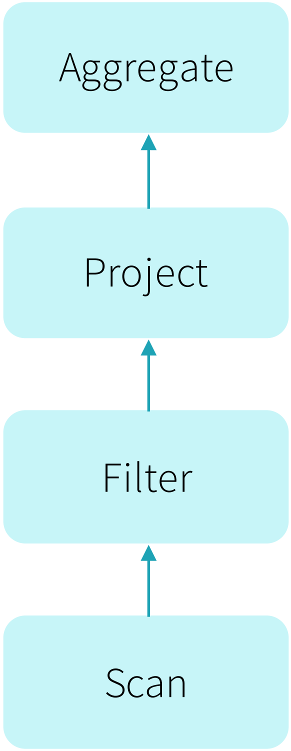
Going back to the fundamentals

Difficult to get order of magnitude performance speed ups with profiling techniques

- For 10x improvement, would need of find top hotspots that add up to 90% and make them instantaneous
- For 100x, 99%

Instead, look bottom up, how fast should *it* run?

```sql
select count(*) from store_sales
where ss_item_sk = 1000
```



Volcano Iterator Model

Standard for 30 years: almost all databases do it

Each operator is an "iterator" that consumes records from its input operator

```
class Filter(
    child: Operator,
    predicate: (Row => Boolean))
  extends Operator {
  def next(): Row = {
  var current = child.next()
  while (current == null ||predicate(current)) {
    current = child.next()
  }
  return current
  }
}
```

Downside of the Volcano Model

1. Too many virtual function calls
   - at least 3 calls for each row in Aggregate

2. Extensive memory access
   - "row" is a small segment in memory (or in L1/L2/L3 cache)

3. Can't take advantage of modern CPU features
   - SIMD, pipelining, prefetching, branch prediction, ILP, instruction cache, …

What if we hire a college freshman to implement this query in Java in 10 mins?

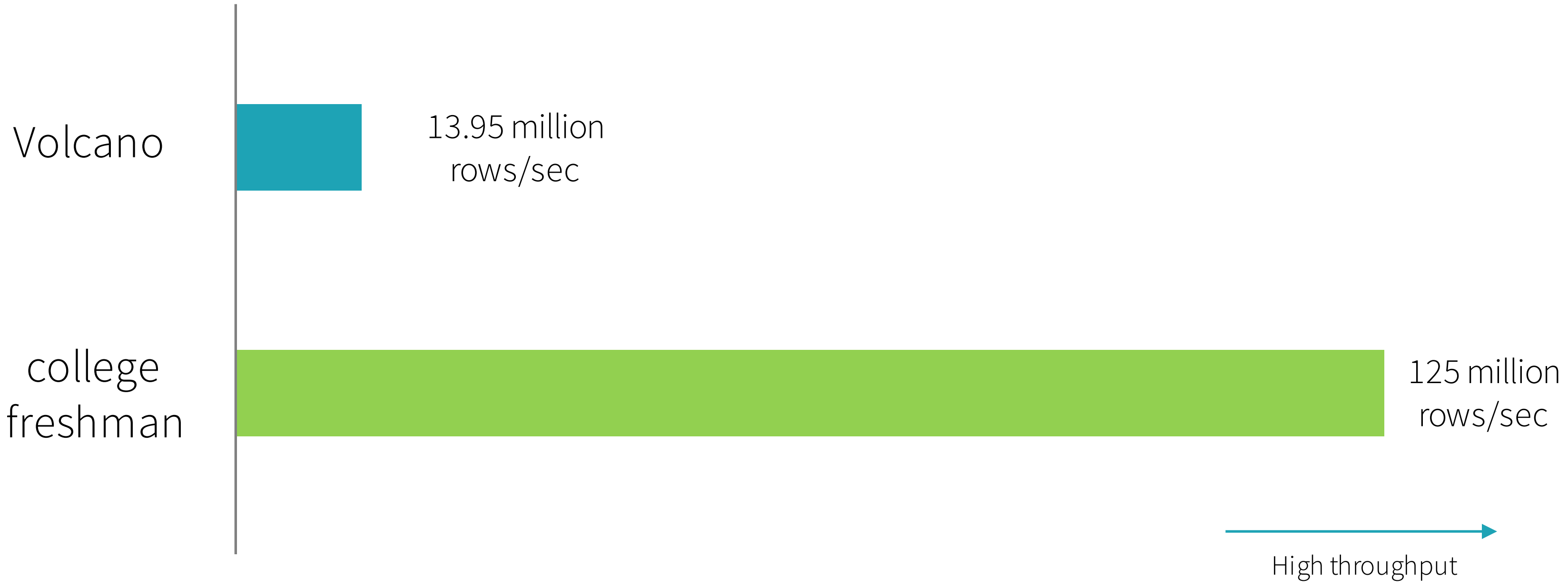
```
select count(*) from store_sales
where ss_item_sk = 1000
```

```
long count = 0;
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1;
  }
}
```

Volcano model
30+ years of database research

vs

college freshman
hand-written code in 10 mins



Note: End-to-end, single thread, single column, and data originated in Parquet on disk

How does a student beat 30 years of research?

Volcano

1. Many virtual function calls
2. Data in memory (or cache)
3. No loop unrolling, SIMD, pipelining

hand-written code

1. No virtual function calls
2. Data in CPU registers
3. Compiler loop unrolling, SIMD, pipelining

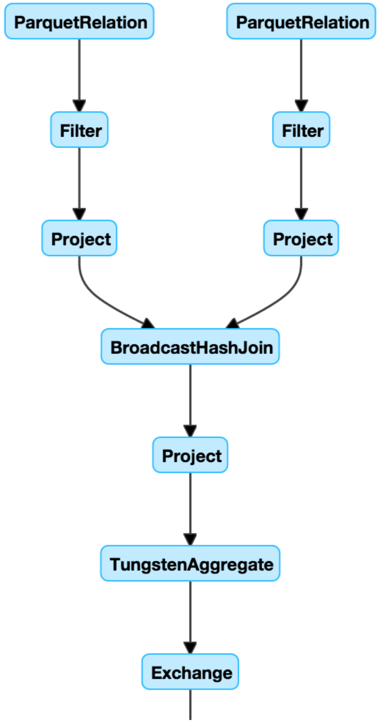
Take advantage of all the information that is known **after** query compilation

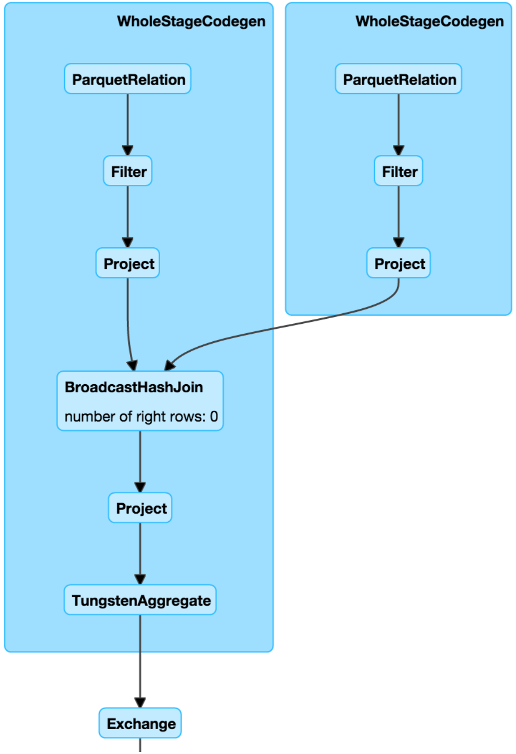
Whole-stage Codegen

Fusing operators together so the generated code looks like hand optimized code:

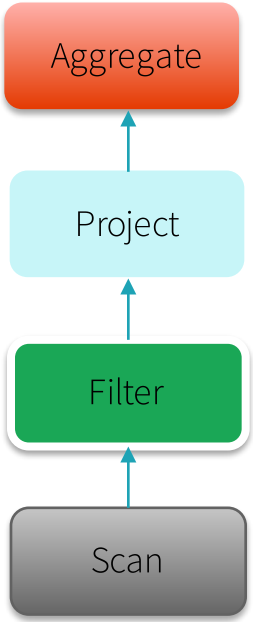
- Identify chains of operators ("stages")
- Compile each stage into a single function
- Functionality of a general purpose execution engine; performance as if hand built system just to run your query

Whole-stage Codegen: Planner





Whole-stage Codegen: Spark as a "Compiler"




```
long count = 0;
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1;
  }
}
```

But there are things we can't fuse

Complicated I/O

- CSV, Parquet, ORC, …
- Sending across the network

External integrations

- Python, R, scikit-learn, TensorFlow, etc
- Reading cached data

Columnar in memory format

In-memory Row Format

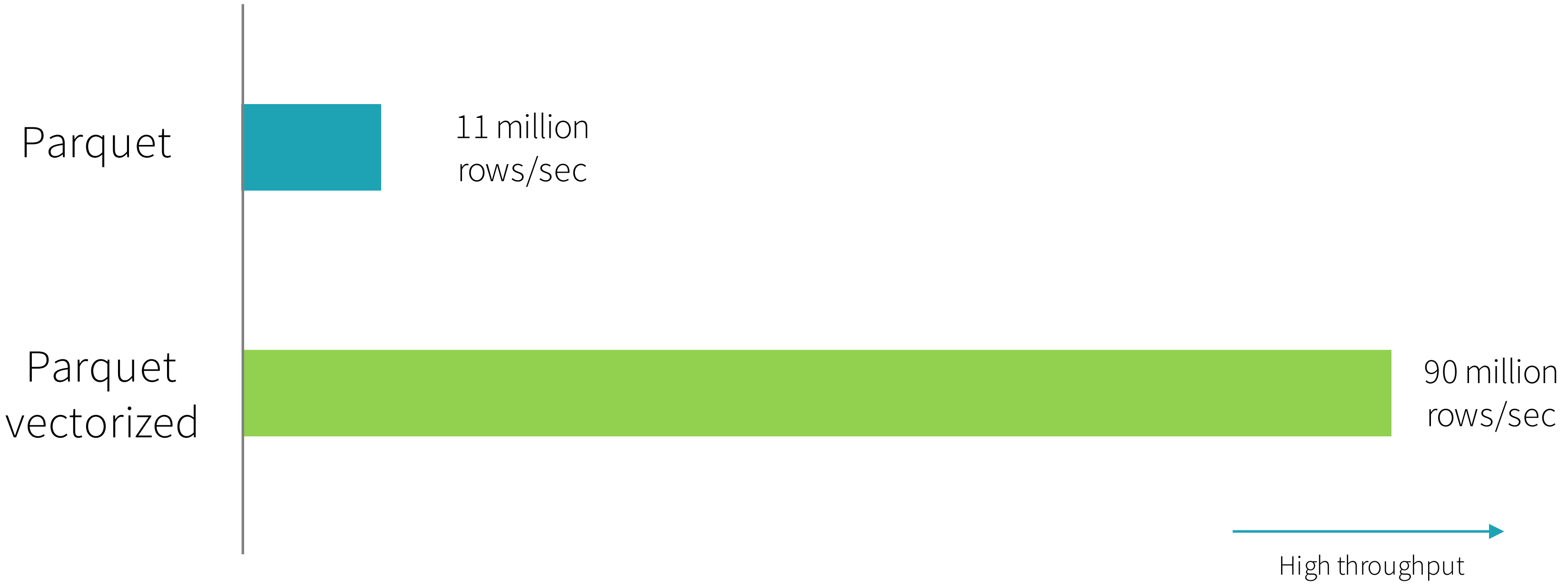
1	john	4.1
2	mike	3.5
3	sally	6.4

In-memory Column Format

1	2	3
john	mike	sally
4.1	3.5	6.4

Why columnar?

1. More efficient: denser storage, regular data access, easier to index into. Enables vectorized processing.
2. More compatible: Most high-performance external systems are already columnar (numpy, TensorFlow, Parquet); zero serialization/copy to work with them
3. Easier to extend: process encoded data, integrate with columnar cache etc.



Note: End-to-end, single thread, single column, and data originated in Parquet on disk

Putting it All Together

Phase 1

Spark 1.4 - 1.6

Memory Management

Code Generation

Cache-aware Algorithms

Phase 2

Spark 2.0+

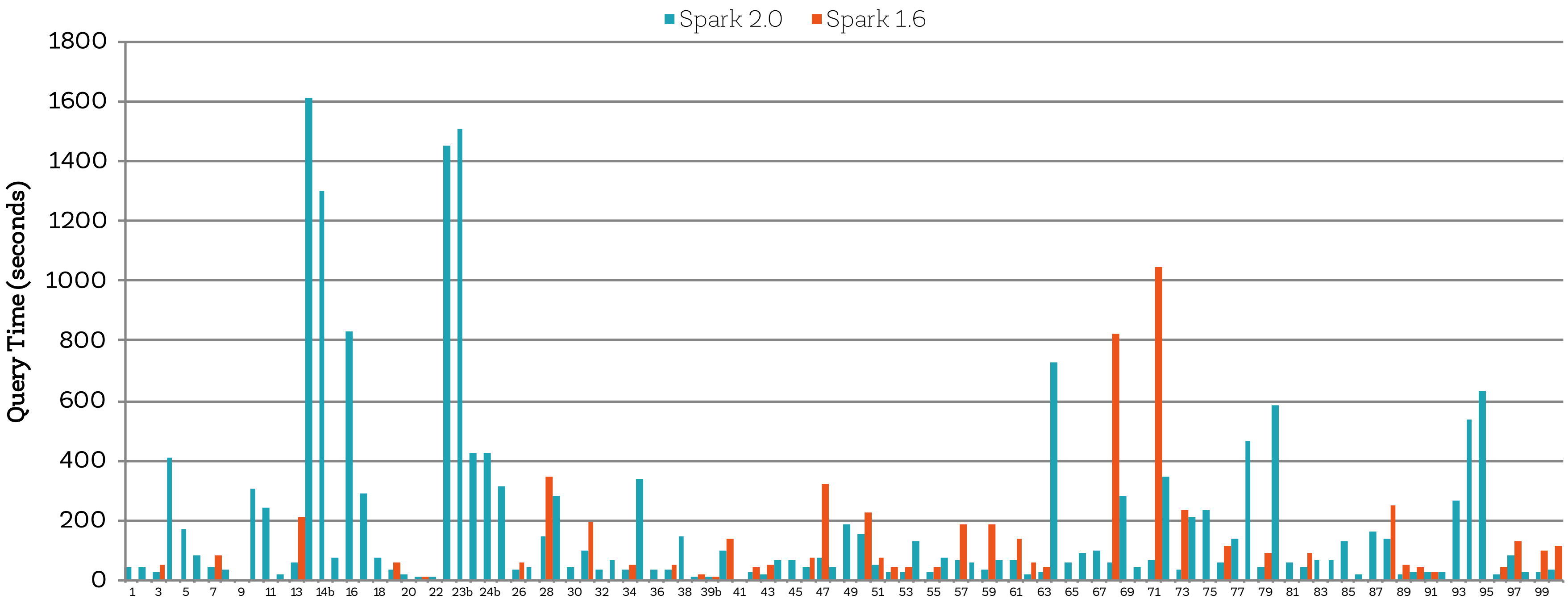
Whole-stage Code Generation

Columnar in Memory Support

Operator Benchmarks: Cost/Row (ns)

primitive	Spark 1.6	Spark 2.0
filter	15 ns	1.1 ns
sum w/o group	14 ns	0.9 ns
sum w/ group	79 ns	10.7 ns
hash join	115 ns	4.0 ns
sort (8-bit entropy)	620 ns	5.3 ns
sort (64-bit entropy)	620 ns	40 ns
sort-merge join	750 ns	700 ns
Parquet decoding (single int column)	120 ns	13 ns

TPC-DS (Scale Factor 1500, 100 cores)



databricks

Status

- Being released as part of Spark 2.0
  - Both Whole stage codegen and vectorized Parquet reader is on by default
- Back to profiling techniques
  - Improve quality of generated code, optimize Parquet reader further
- Try it out and let us know!

Further Reading

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

Spark Summit 2016 will be held in San Francisco on June 6–8. Check out the full agenda and get your ticket before it sells out!

---

http://tinyurl.com/project-tungsten

2016 Apache Spark Survey

Spark Summit EU
Brussels
October 25-27

The CFP closes at **11:59pm on July 1st**
For more information and to submit:

https://spark-summit.org/eu-2016/

Questions?

databricks™