Apache Spark's Performance
Project Tungsten and Beyond

Sameer Agarwal

Spark Summit| Brussels| Oct 26th 2016

databricks

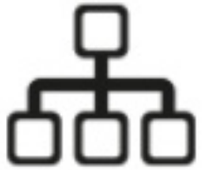
databricks

About Me

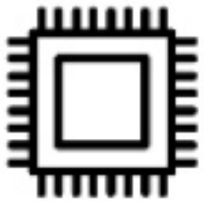
- Software Engineer at Databricks (Spark Core/SQL)
- PhD in Databases (AMPLab, UC Berkeley)
- Research on BlinkDB (Approximate Queries in Spark)

Hardware Trends

Storage

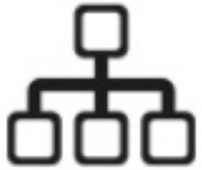
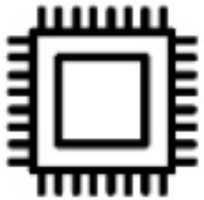


Network

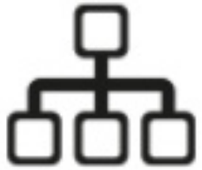
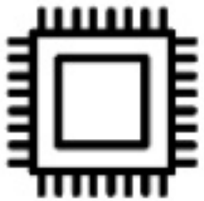


CPU

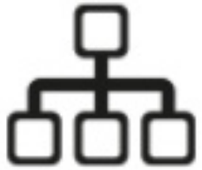
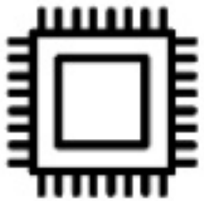
Hardware Trends

	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)

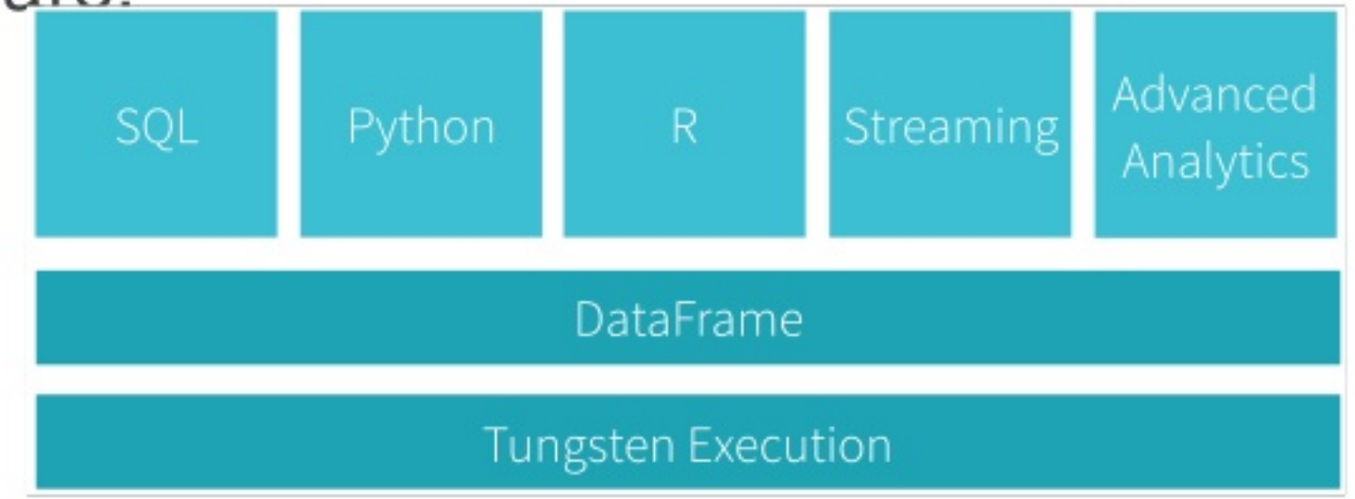
Data formats have improved

- 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”: relatively easy to pick broadcast hash 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 explicit memory management instead of relying on Java objects

- Reduce memory footprint
- Eliminate garbage collection overheads
- Use sun.misc.unsafe rows 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

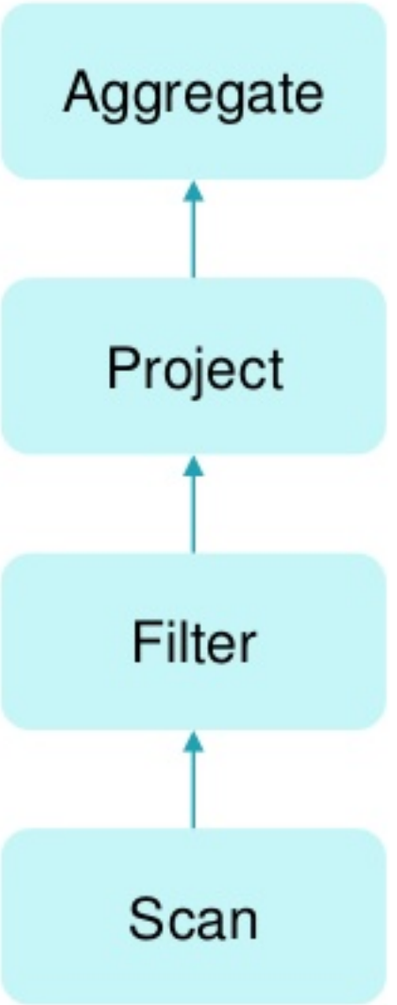
Going back to the fundamentals

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?

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



Volcano—An Extensible and Parallel Query Evaluation System

Goetz Graefe

Abstract—**To investigate the interactions of extensibility and parallelism in database query processing, we have developed a new dataflow query execution system called Volcano. The Volcano effort provides a rich environment for research and education in database systems design, heuristics for query optimization, parallel query execution, and resource allocation.**

Volcano uses a standard interface between algebra operators, allowing easy addition of new operators and operator implementations. Operations on individual items, e.g., predicates, are imported into the query processing operators using *support functions*. The semantics of support functions is not prescribed; any data type including complex objects and any operation can be realized. Thus, Volcano is *extensible* with new operators, algorithms, data types, and type-specific methods.

Volcano includes two novel *meta-operators*. The *choose-plan*

tem as it lacks features such as a user-friendly query language, a type system for instances (record definitions), a query optimizer, and catalogs. Because of this focus, Volcano is able to serve as an experimental vehicle for a multitude of purposes, all of them open-ended, which results in a combination of requirements that have not been integrated in a single system before. First, it is modular and extensible to enable future research, e.g., on algorithms, data models, resource allocation, parallel execution, load balancing, and query optimization heuristics. Thus, Volcano provides an infrastructure for experimental research rather than a final research prototype in itself. Second, it

G. Graefe, **Volcano— An Extensible and Parallel Query Evaluation System**, *In IEEE Transactions on Knowledge and Data Engineering 1994*

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
  }
}
```

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

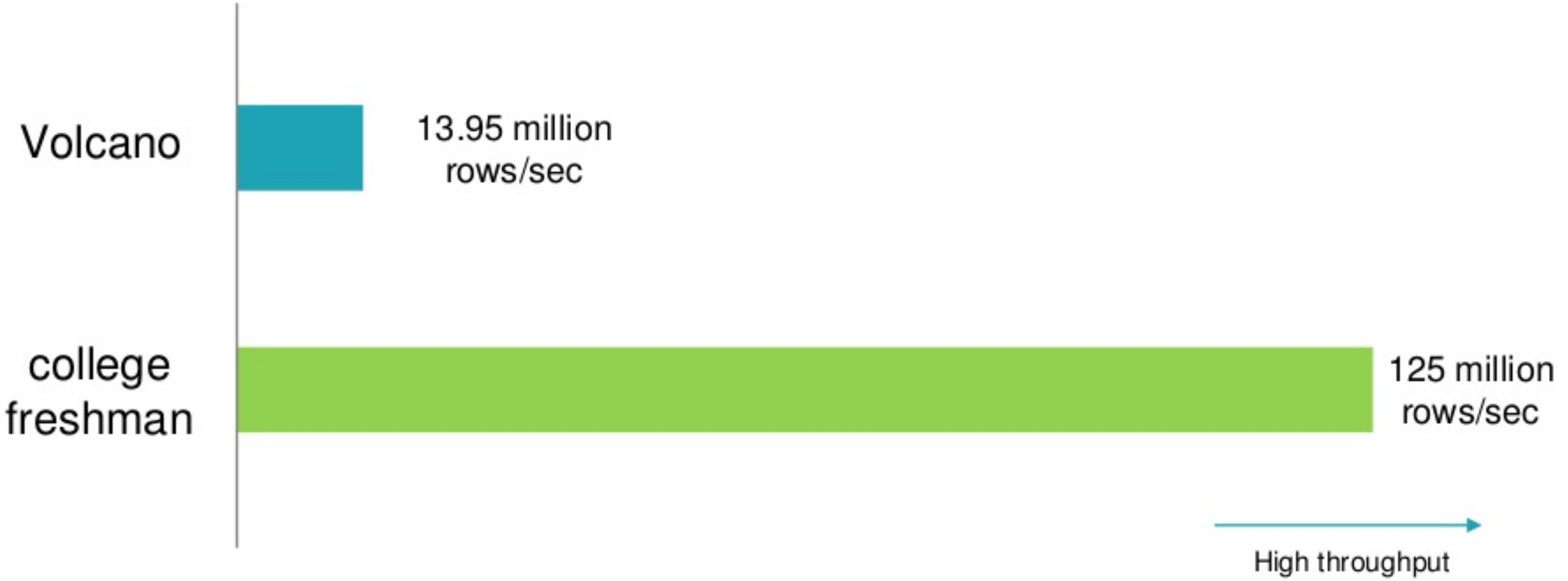
```
select count(*) from store_sales
where ss_item_sk = 1000
```

```
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



databricks

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 Model

1. Too many virtual function calls
2. Intermediate data in memory (or L1/L2/L3 cache)
3. Can't take advantage of modern CPU features -- no loop unrolling, SIMD, pipelining, prefetching, branch prediction etc.

Hand-written code

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

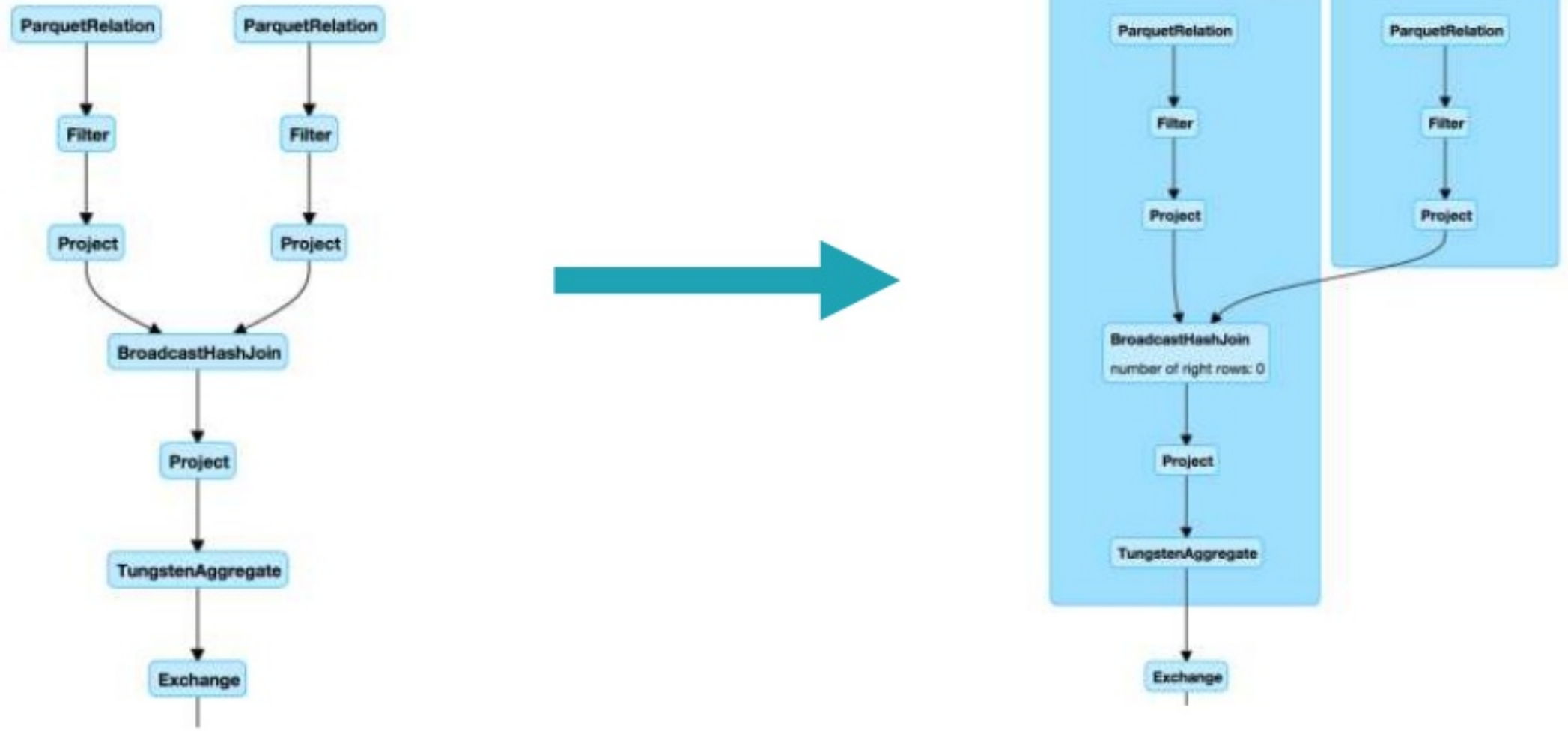
Take advantage of all the information that is known after query compilation

Whole-stage Codegen

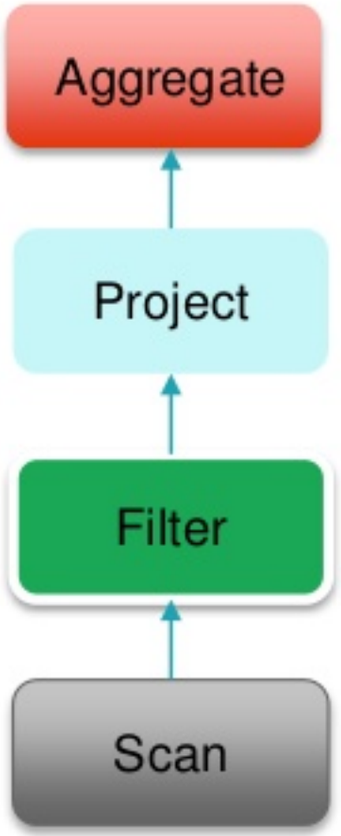
Fusing operators together so the generated code looks like hand optimized code:

- 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”



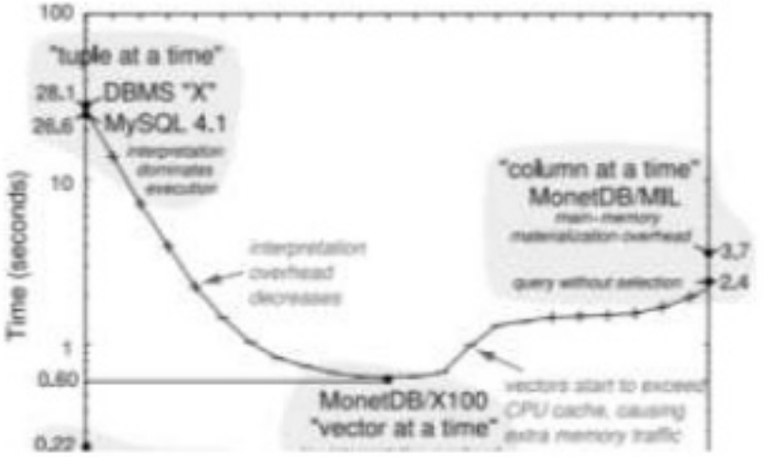
```
long count = 0;
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1;
  }
}
```

Efficiently Compiling Efficient Query Plans for Modern Hardware

Thomas Neumann
Technische Universität München
Munich, Germany
neumann@in.tum.de

ABSTRACT

As main memory grows, query performance is more and more determined by the raw CPU costs of query processing itself. The classical iterator style query processing technique is very simple and flexible, but shows poor performance on modern CPUs due to lack of locality and frequent instruction mispredictions. Several techniques like batch oriented processing or vectorized tuple processing have been proposed in the past to improve this situation, but even these techniques are



T Neumann, Efficiently compiling efficient query plans for modern hardware. *In VLDB 2011*

But there are things we can't fuse

Complicated I/O

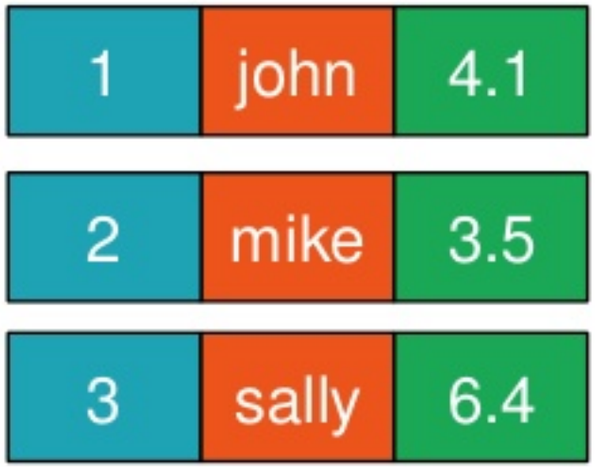
- CSV, Parquet, ORC, …
- Sending across the network

External integrations

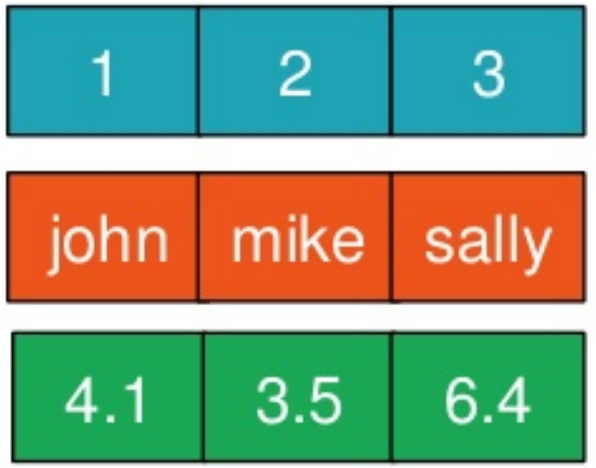
- Python, R, scikit-learn, TensorFlow, etc

Columnar in memory format

In-memory
Row Format

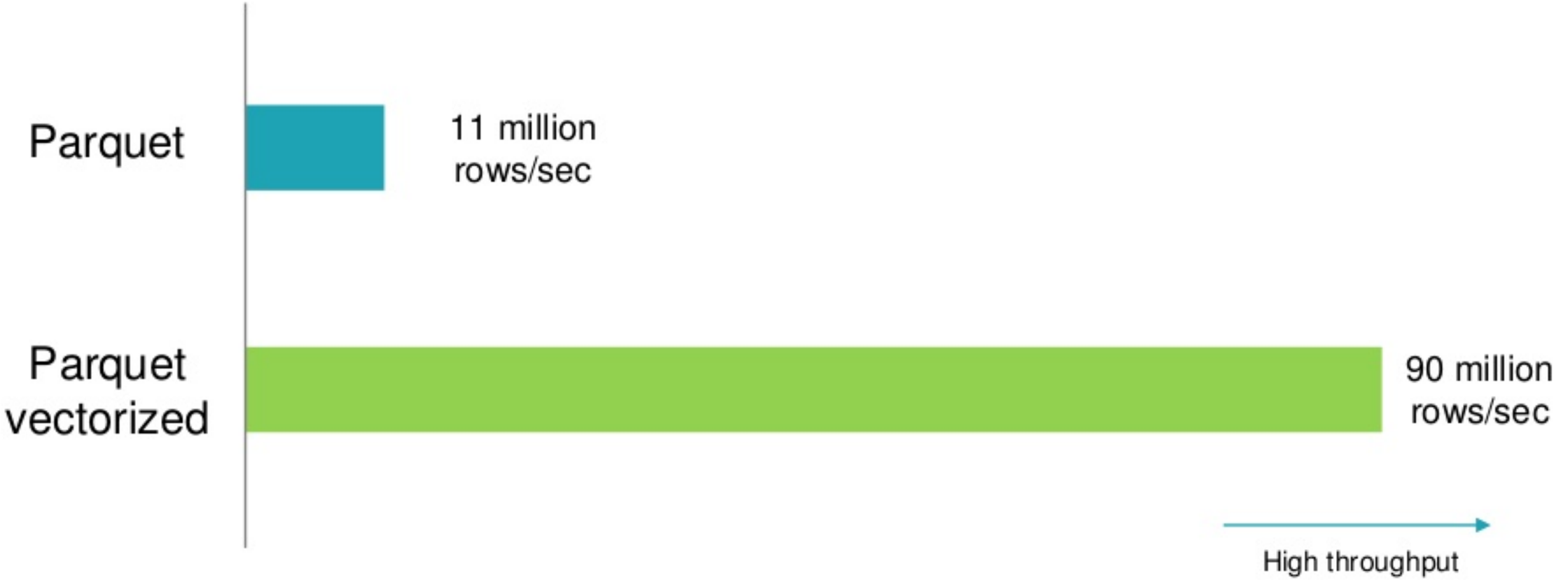


In-memory
Column Format



Why columnar?

1. More efficient: denser storage, regular data access, easier to index into
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



databricks

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

Putting it All Together

Demo

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

Both whole stage codegen [SPARK-12795] and the vectorized parquet reader [SPARK-12992] are enabled by default in Spark 2.0+

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

5-30x Speedups

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

Radix Sort
10-100x
Speedups

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

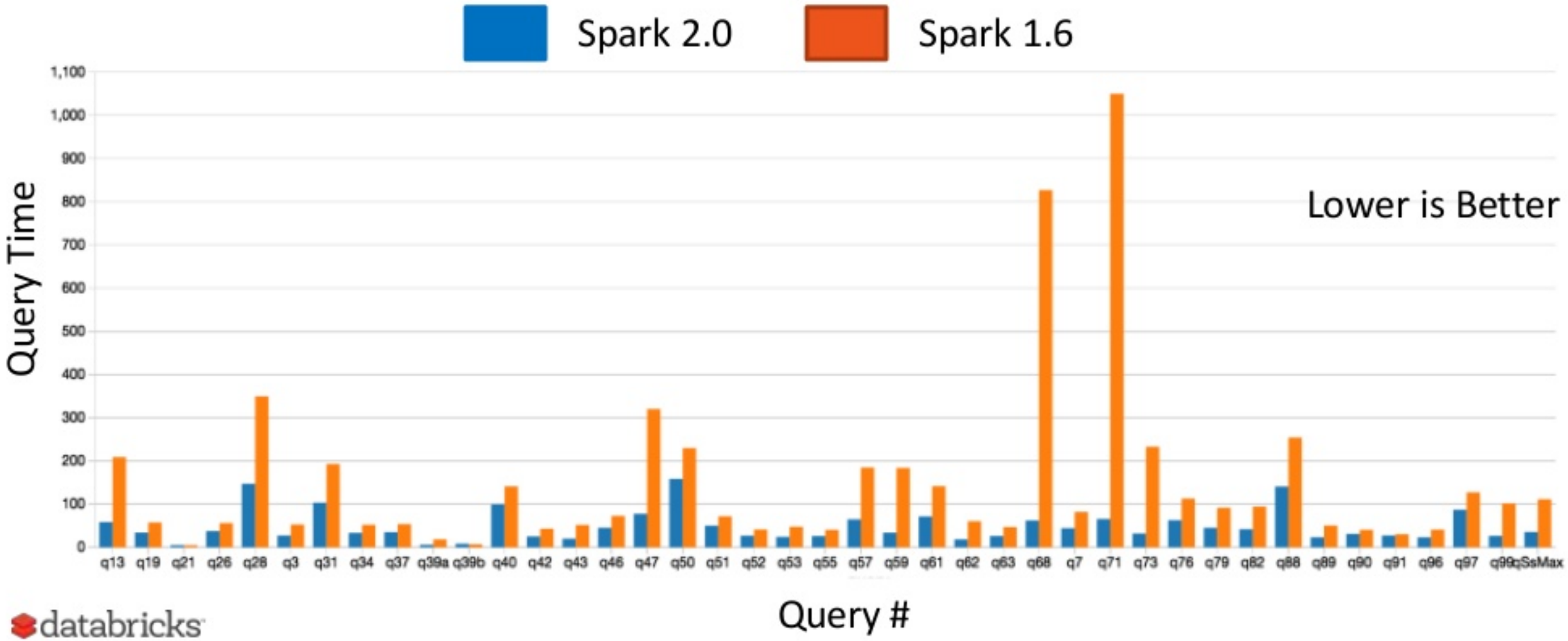
Shuffling still the bottleneck

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

10x Speedup

TPC-DS (Scale Factor 1500, 100 cores)



databricks

What’s Next?

Spark 2.1, 2.2 and beyond

1. SPARK-16026: Cost Based Optimizer
   - Leverage table/column level statistics to optimize joins and aggregates
   - Statistics Collection Framework (Spark 2.1)
   - Cost Based Optimizer (Spark 2.2)
2. Boosting Spark's Performance on Many-Core Machines
   - Qifan's Talk Today at 2:55pm (Research Track)
   - In-memory/ single node shuffle
3. Improving quality of generated code and better integration with the in-memory column format in Spark

Questions?

I'll also be at the Databricks booth tomorrow from 12-2pm!

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

http://tinyurl.com/project-tungsten