Exceptions are the Norm
Dealing with Bad Actors in ETL

Sameer Agarwal

Spark Summit | Boston | Feb 9th 2017

databricks

About Me

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

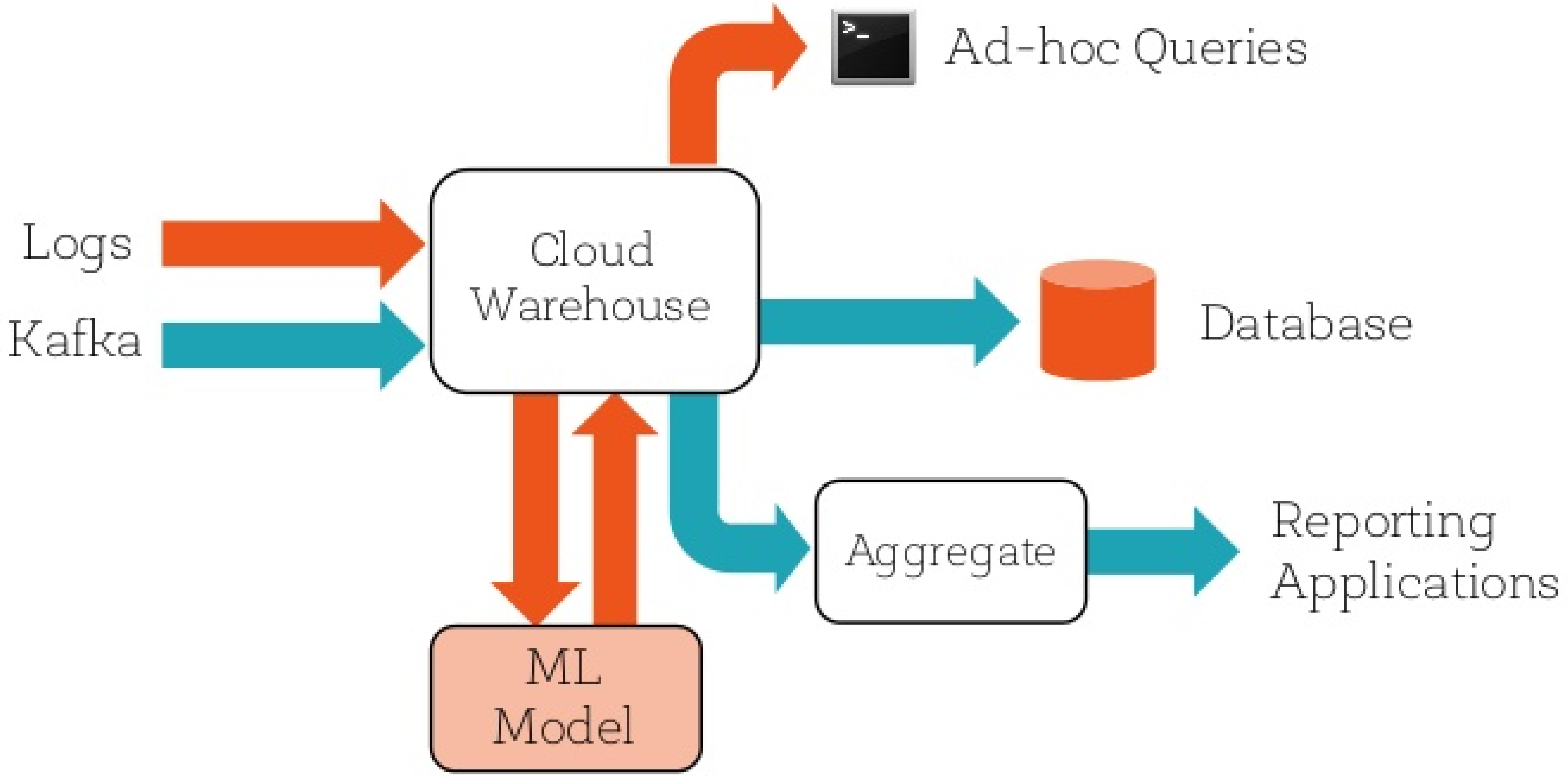
Overview

1. What's an ETL Pipeline?
    - How is it different from a regular query execution pipeline?
2. Using SparkSQL for ETL
    - Dealing with Dirty Data (Bad Records or Files)
    - Performance (Project Tungsten)
3. New Features in Spark 2.2 and 2.3
    - Focus on building ETL-friendly pipelines

What is a Data Pipeline?

1. Sequence of transformations on data
2. Source data is typically semi-structured/unstructured (JSON, CSV etc.)
3. Output data is structured and ready for use by analysts and data scientists
4. Source and destination are often on different storage systems.

Example of a Data Pipeline



databricks

ETL is the First Step in a Data Pipeline

1. ETL stands for `EXTRACT`, `TRANSFORM` and `LOAD`

2. Goal is to "clean" or "curate" the data
   - Retrieve data from source `(EXTRACT)`
   - Transform data into a consumable format `(TRANSFORM)`
   - Transmit data to downstream consumers `(LOAD)`

An ETL Query in Spark

```
spark.read.csv("/source/path")
```

EXTRACT

An ETL Query in Spark

```
spark.read.csv("/source/path")
    .filter(...)
    .agg(...)
```

EXTRACT

TRANSFORM

An ETL Query in Spark

```
spark.read.csv("/source/path")
    .filter(...)
    .agg(...)
    .write.mode("append")
    .parquet("/output/path")
```

EXTRACT

TRANSFORM

LOAD

What's so hard about ETL Queries?

Why is ETL Hard?

1. Data can be Messy
   - Incomplete information
   - Missing data stored as empty strings, “none”, “missing”, “xxx” etc.
2. Data can be Inconsistent
   - Data conversion and type validation in many cases is error-prone
     - For e.g., expecting a number but found ”123 000”
     - different formats “31/12/2017” “12/31/2017”
   - Incorrect information
     - For e.g., expecting 5 fields in CSV, but can’t find 5 fields.

Why is ETL Hard?

3. Data can be Constantly Arriving
    - At least once or exactly once semantics
    - Fault tolerance
    - Scalability

4. Data can be Complex
    - For e.g., Nested JSON data to extract and flatten
    - Dealing with inconsistency is even worse

This is why ETL is important

Consumers of this data don't want to deal with this messiness and complexity

On the flip side

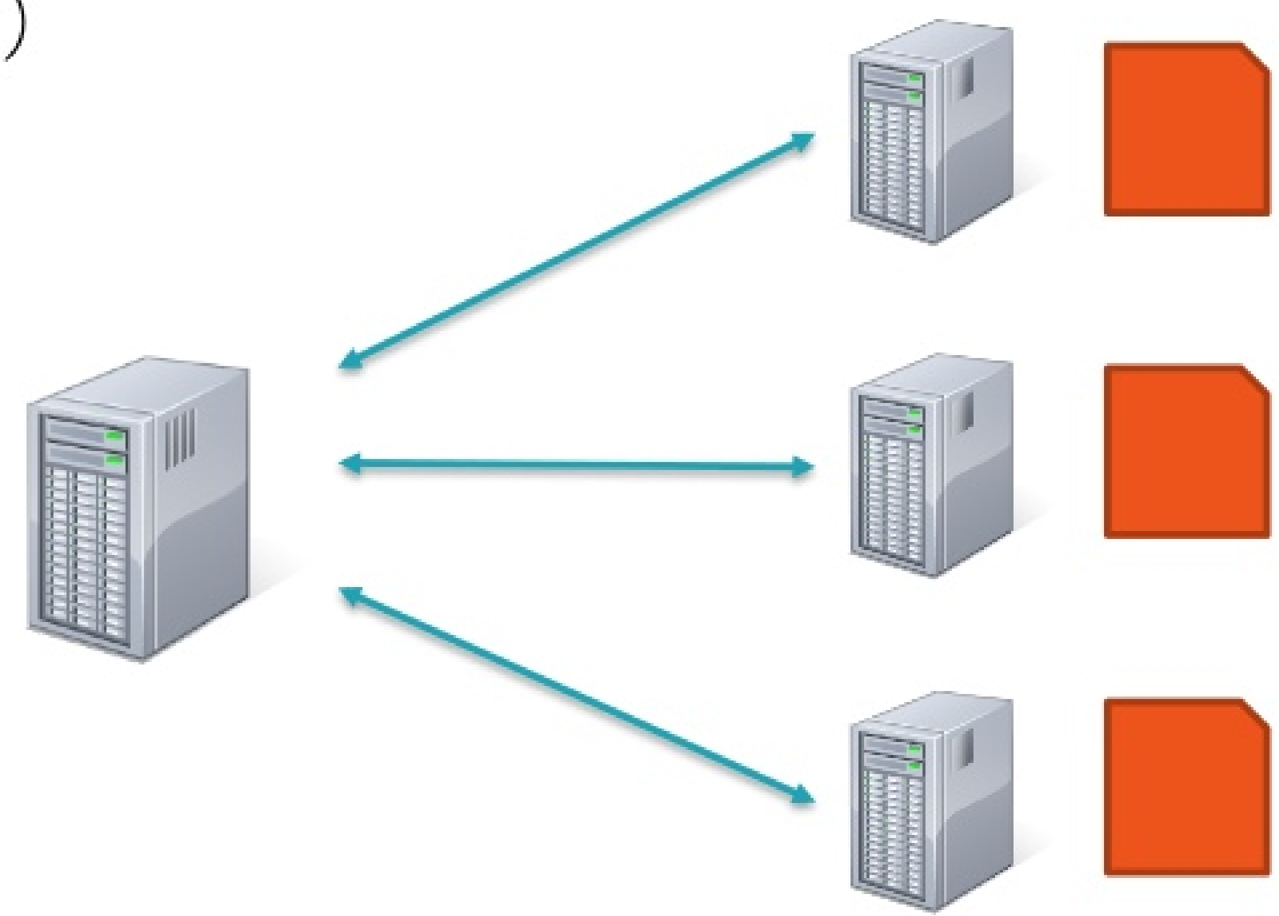
1. A few bad records can fail a job
   - These are not the same as transient errors
   - No recourse for recovery
2. Support for ETL features
   - File formats and conversions have gaps
   - For e.g., multi-line support, date conversions
3. Performance

Spark's flexible APIs, support for a wide variety of datasources and state of art tungsten execution engine makes it a great framework for building end-to-end ETL Pipelines

Using SparkSQL for ETL

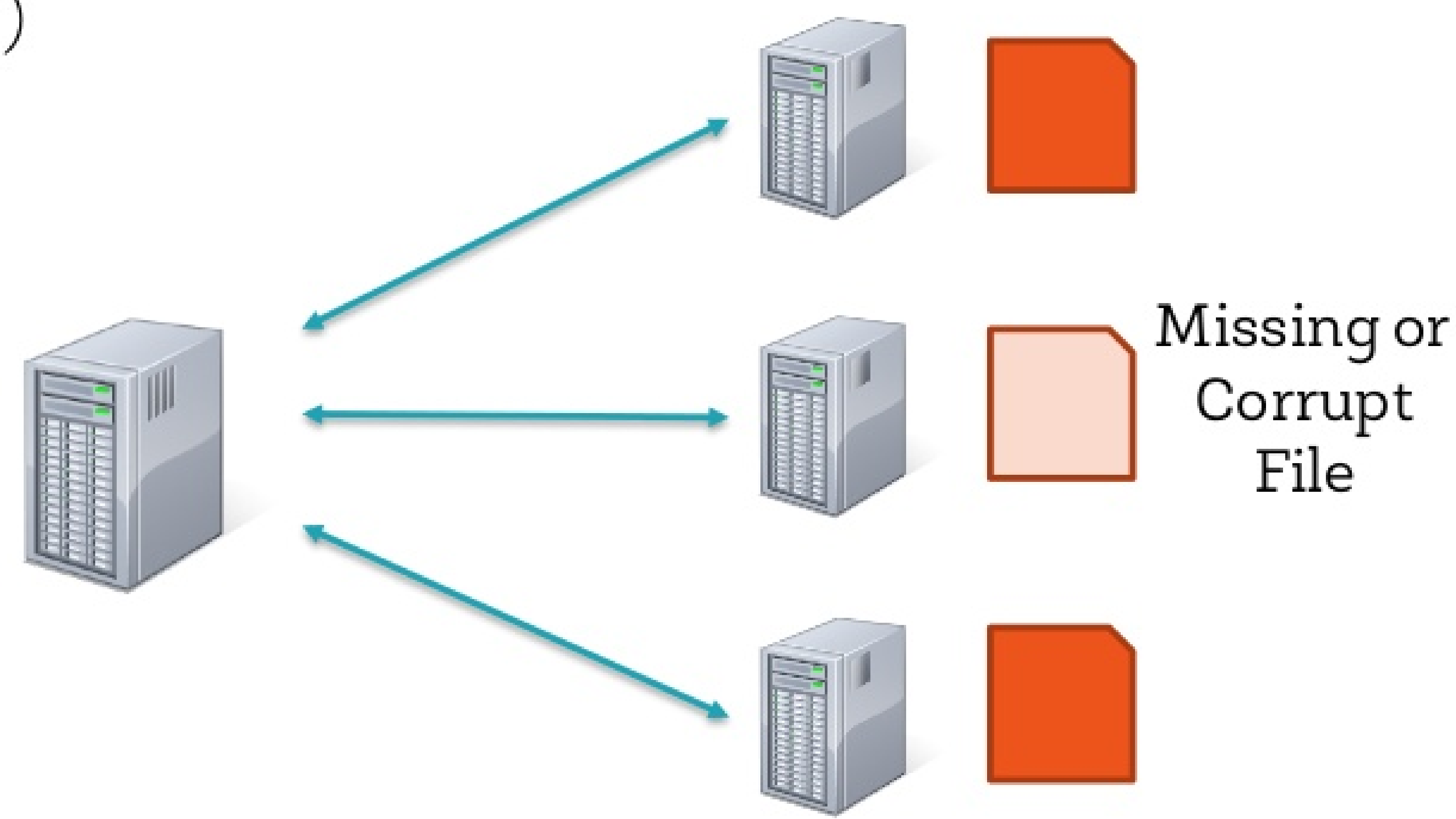
Dealing with Bad Data: Skip Corrupt Files

```
spark.read.csv("/source/path")
      .filter(...)
      .agg(...)
      .write.mode("append")
      .parquet("/output/path")
```



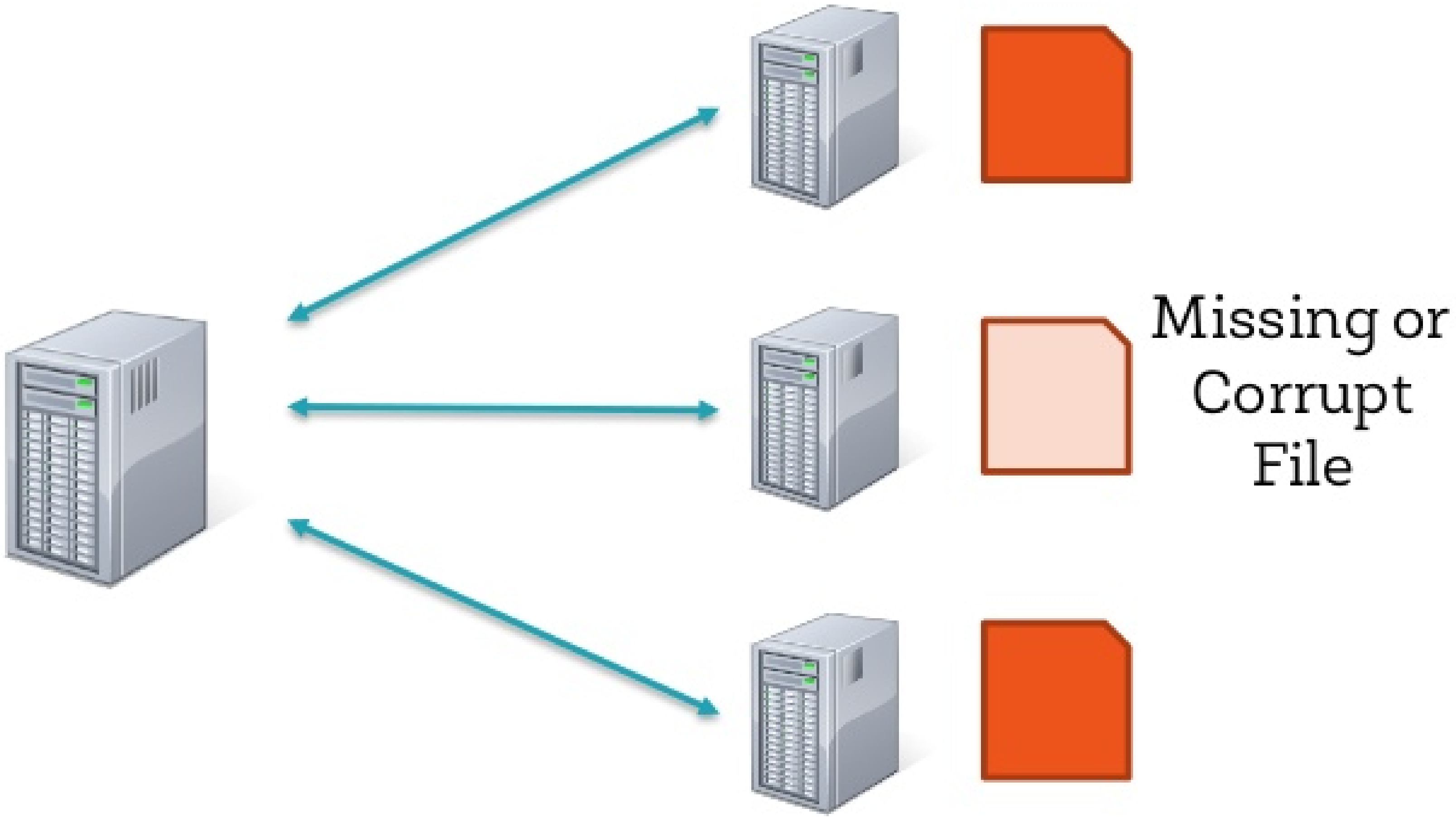
Dealing with Bad Data: Skip Corrupt Files

```
spark.read.csv("/source/path")
      .filter(...)
      .agg(...)
      .write.mode("append")
      .parquet("/output/path")
```



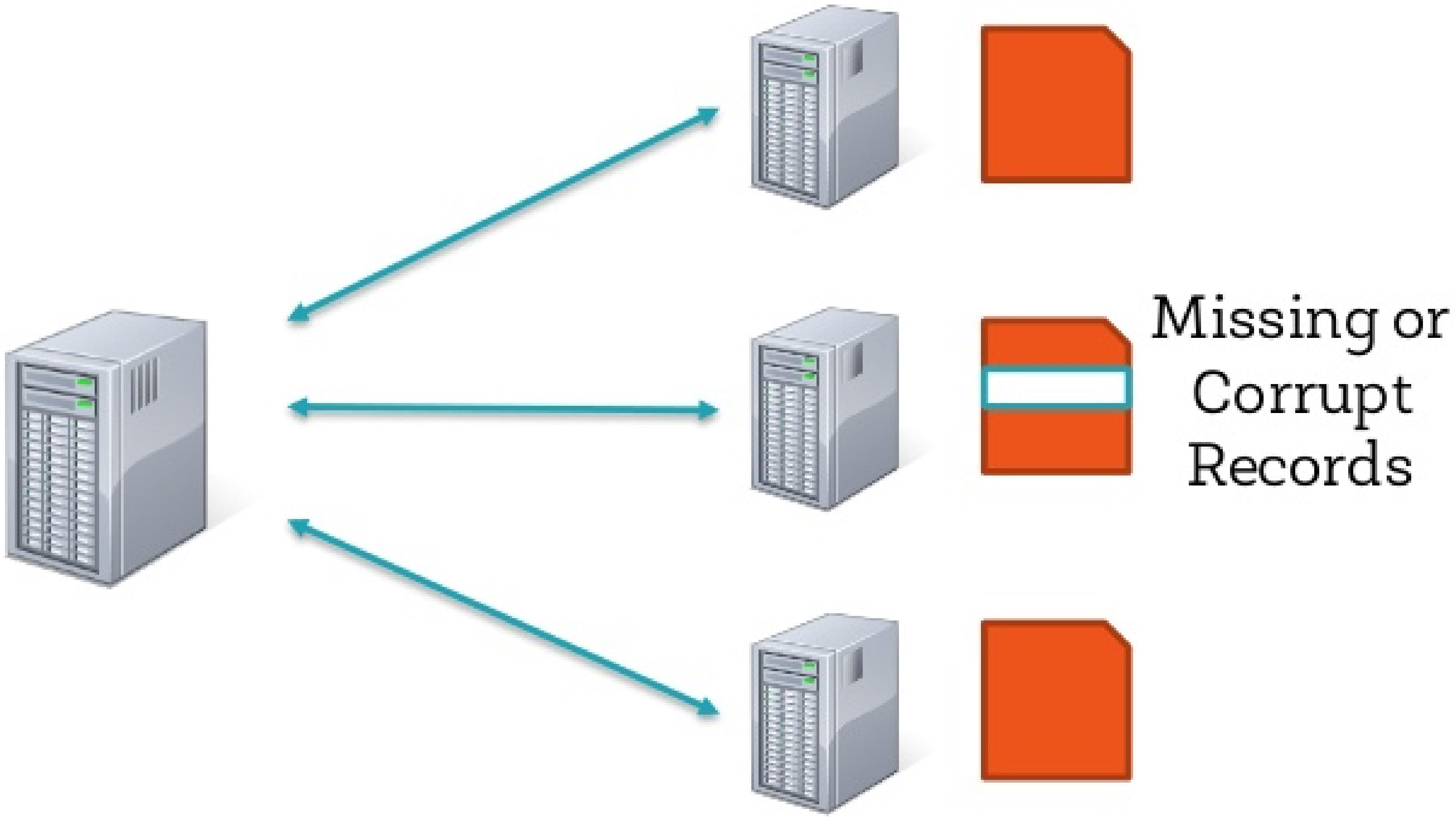
Dealing with Bad Data: Skip Corrupt Files

[SPARK-17850] If true, the Spark jobs will continue to run even when it encounters corrupt or non-existent files. The contents that have been read will still be returned.



spark.sql.files.ignoreCorruptFiles = true

Dealing with Bad Data: Skip Corrupt Records

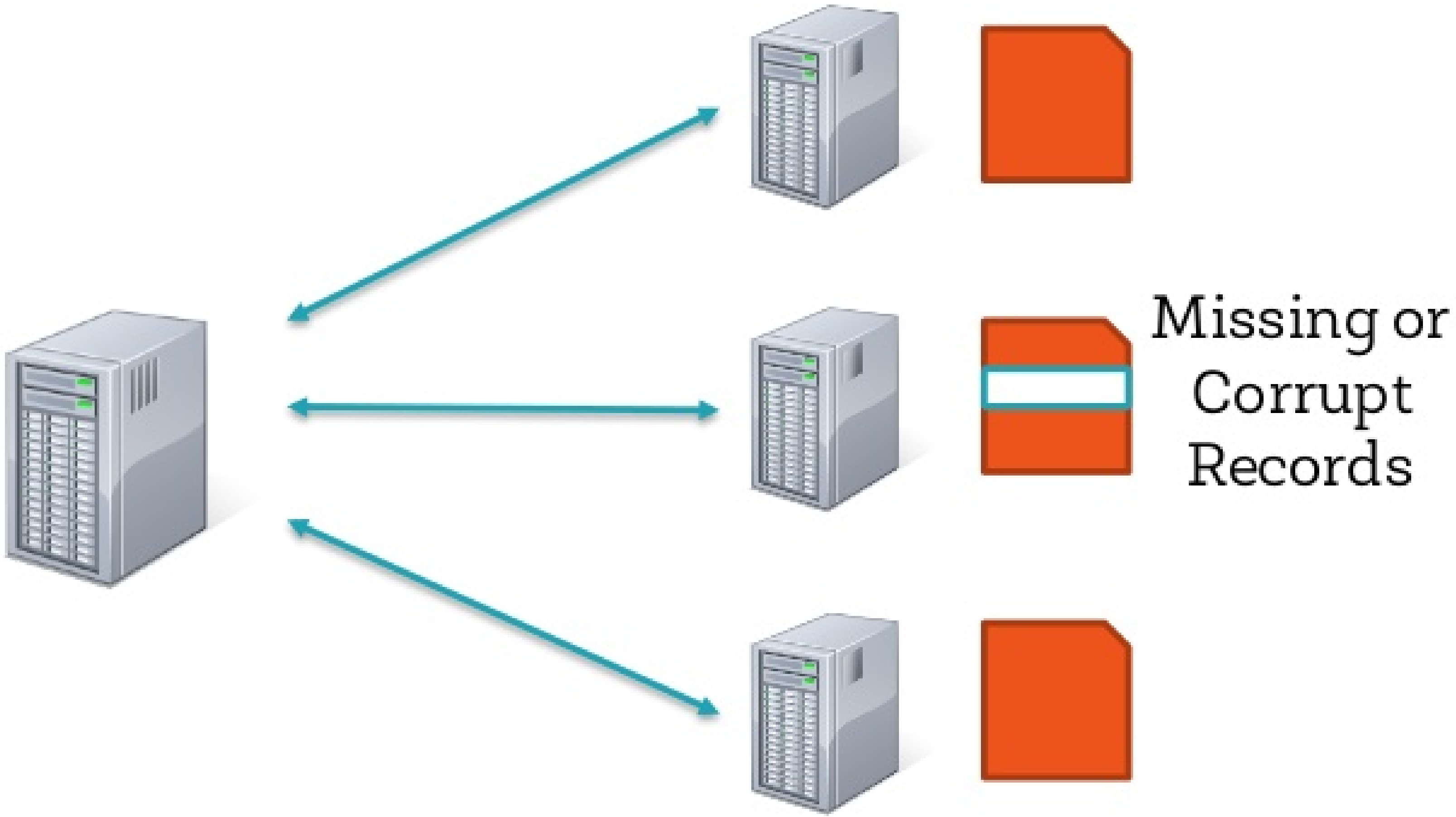


databricks

Dealing with Bad Data: Skip Corrupt Records

[SPARK-12833][SPARK-13764] TextFile formats (JSON and CSV) support 3 different ParseModes while reading data:

1. PERMISSIVE
2. DROPMALFORMED
3. FAILFAST



JSON: Dealing with Corrupt Records

```
{"a":1, "b":2, "c":3}
{"a":{, b:3}
{"a":5, "b":6, "c":7}
```

```
spark.read
  .option("mode", "PERMISSIVE")
  .json(corruptRecords)
  .show()
```

Can be configured via
spark.sql.columnNameOfCorruptRecord

```
+---------------+----+----+----+
|_corrupt_record|   a|   b|   c|
+---------------+----+----+----+
|           null|   1|   2|   3|
|   {"a":{, b:3}|null|null|null|
|           null|   5|   6|   7|
+---------------+----+----+----+
```

JSON: Dealing with Corrupt Records

```
{"a":1, "b":2, "c":3}
```
~~{"a":{, b:3}~~
```
{"a":5, "b":6, "c":7}
```

```
spark.read
  .option("mode", "DROPMALFORMED")
  .json(corruptRecords)
  .show()
```

```
+---+---+---+
|  a|  b|  c|
+---+---+---+
|  1|  2|  3|
|  5|  6|  7|
+---+---+---+
```

JSON: Dealing with Corrupt Records

```
{"a":1, "b":2, "c":3}
{"a":{, b:3}
{"a":5, "b":6, "c":7}
```

```
spark.read
  .option("mode", "FAILFAST")
  .json(corruptRecords)
  .show()
```

```
org.apache.spark.sql.catalyst.json
.SparkSQLJsonProcessingException:
Malformed line in FAILFAST mode:
{"a":{, b:3}
```

databricks

CSV: Dealing with Corrupt Records

```
year,make,model,comment,blank
"2012","Tesla","S","No comment",
1997,Ford,E350,"Go get one now they",
2015,Chevy,Volt
```

```
spark.read
  .format("csv")
  .option("mode", "PERMISSIVE")
  .load(corruptRecords)
  .show()
```

```
+----+-----+-----+--------------------+-----+
|year| make|model|             comment|blank|
+----+-----+-----+--------------------+-----+
|2012|Tesla|    S|          No comment| null|
|1997| Ford| E350|Go get one now th...| null|
|2015|Chevy| Volt|                null| null|
+----+-----+-----+--------------------+-----+
```

CSV: Dealing with Corrupt Records

```
year,make,model,comment,blank
"2012","Tesla","S","No comment",
1997,Ford,E350,"Go get one now they",
```
~~2015,Chevy,Volt~~

```
spark.read
   .format("csv")
   .option("mode", "DROPMALFORMED")
   .load(corruptRecords)
   .show()
```

```
+----+-----+-----+--------------------+-----+
|year| make|model|             comment|blank|
+----+-----+-----+--------------------+-----+
|2012|Tesla|    S|          No comment| null|
|1997| Ford| E350|Go get one now th...| null|
+----+-----+-----+--------------------+-----+
```

CSV: Dealing with Corrupt Records

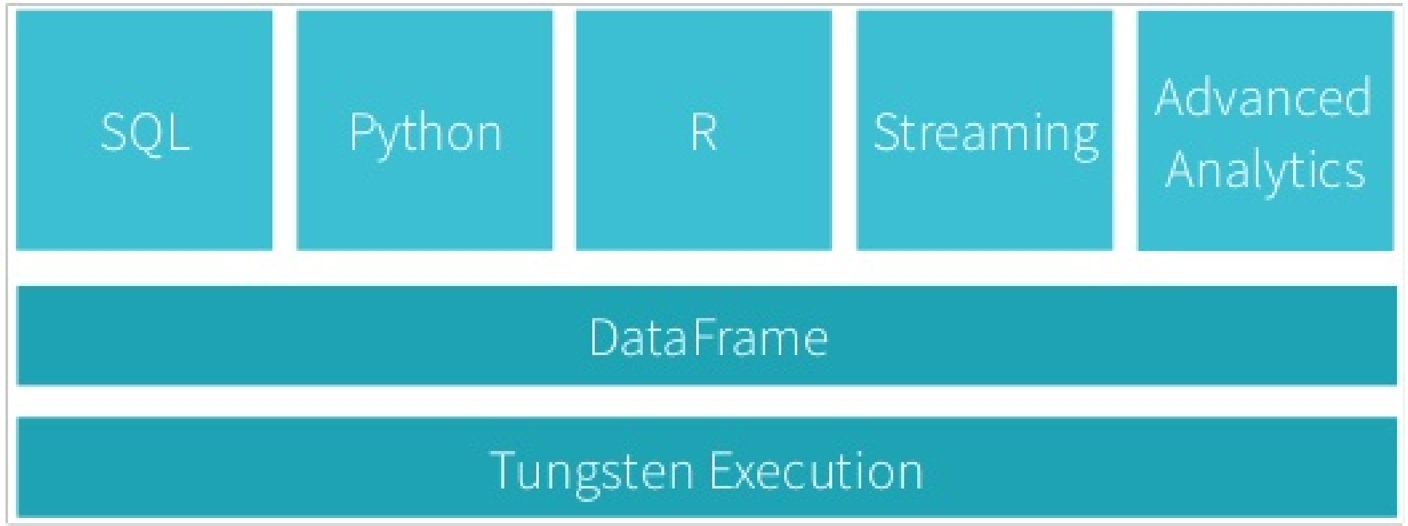
```
year,make,model,comment,blank
"2012","Tesla","S","No comment",
1997,Ford,E350,"Go get one now they",
2015,Chevy,Volt
```

```
spark.read
  .format("csv")
  .option("mode", "FAILFAST")
  .load(corruptRecords)
  .show()
```

```
java.lang.RuntimeException:
Malformed line in FAILFAST mode:
2015,Chevy,Volt
```

Spark Performance: Project Tungsten

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



Spark Performance: Project Tungsten

Phase 1

Foundation

Memory Management

Code Generation

Cache-aware Algorithms

Phase 2

Order-of-magnitude Faster

Whole-stage Codegen

Vectorization

SparkSQL: A Compiler from Queries to RDDs (Developer Track at 5:40pm)

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

databricks

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

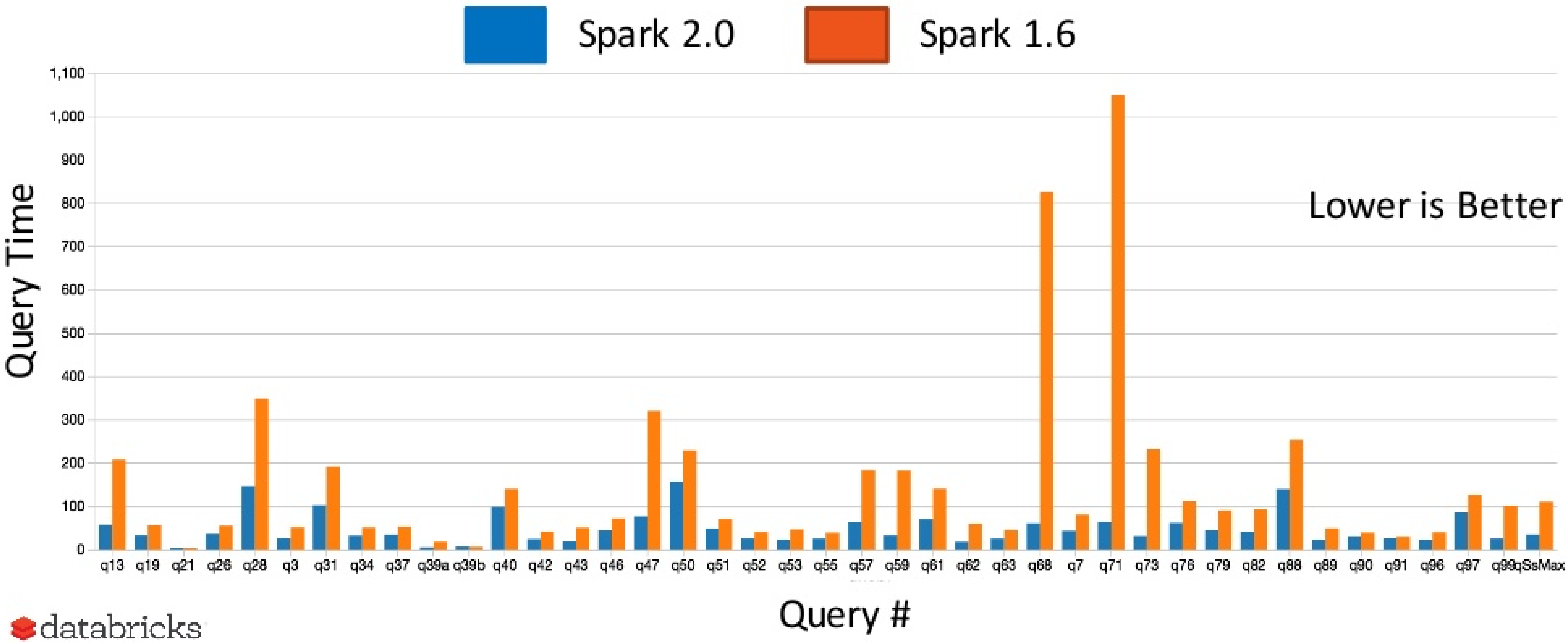
databricks

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

Apache Spark 2.2 and 2.3

Massive focus on building ETL-friendly pipelines

New Features in Spark 2.2 and 2.3

1. Better Functionality:
    - Improved JSON and CSV Support
2. Better Usability:
    - Better Error Messages
3. Better Performance:
    - SQL Execution
    - Python UDF Processing

Functionality: Better JSON Support

1. [SPARK-18352] Multi-line JSON Support
   - Spark currently reads JSON one line at a time
   - This currently requires custom ETL

```
spark.read
  .option("wholeFile",true)
  .json(path)
```

Availability: Spark 2.2

Functionality: Better JSON Support

2. [SPARK-19480] Higher order functions in SQL
   - Enable users to manipulate nested data in Spark
   - Operations include map, filter, reduce on arrays/maps

```
tbl_x
|-- key: long (nullable = false)
|-- values: array (nullable = false)
|    |-- element: long (containsNull = false)
```

Functionality: Better JSON Support

2. [SPARK-19480] Higher order functions in SQL

```
tbl_x
|-- key: long (nullable = false)
|-- values: array (nullable = false)
|    |-- element: long (containsNull = false)
```

```
SELECT key, TRANSFORM(values, v -> v + key)
FROM tbl_x
```

Availability: Spark 2.3+

Functionality: Better CSV Support

1. [SPARK-16099] Improved/Performant CSV Datasource
   - Multiline CSV Support
   - Additional options for CSV Parsing
   - Whole text reader for dataframes

Availability: Spark 2.2

Functionality: Better ETL Support

1. More Fine-grained (record-level) tolerance to errors
   - Provide users with controls on how to handle these errors
     - Ignore and report errors post-hoc
     - Ignore bad rows up to a certain number or percentage

Availability: Spark 2.2

Usability: Better Error Messages

1. Spark must explain why data is bad
2. This is especially true for data conversion
   - `scala.MatchError: start (of class java.lang.String)`
3. Which row in your source data could not be converted ?
4. Which column could not be converted ?

Availability: Spark 2.2 and 2.3

Performance: SQL Execution

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
   - In-memory/ single node shuffle
3. Improving quality of generated code and better integration with the in-memory column format in Spark

Performance: Python UDFs

1. Python is the most popular language for ETL
2. Python UDFs are often used to express elaborate data conversions/transformations
3. Any improvements to python UDF processing will ultimately improve ETL.
4. **Next talk:** Improving Python and Spark Performance and Interoperability (Wes McKinney)

Availability: Spark 2.3+

Recap

1. What's an ETL Pipeline?
   - How is it different from a regular query execution pipeline?
2. Using SparkSQL for ETL
   - Dealing with Dirty Data (Bad Records or Files)
   - Performance (Project Tungsten)
3. New Features in Spark 2.2 and 2.3
   - Focus on building ETL-friendly pipelines

Questions?