Building Robust ETL Pipelines with Apache Spark

Xiao Li

Spark Summit | SF | Jun 2017

databricks

About Databricks

TEAM

Started Spark project (now Apache Spark) at UC Berkeley in 2009

MISSION

Making Big Data Simple

PRODUCT

Unified Analytics Platform

About Me

- Apache Spark Committer
- Software Engineer at Databricks
- Ph.D. in University of Florida
- Previously, IBM Master Inventor, QRep, GDPS A/A and STC
- Spark SQL, Database Replication, Information Integration
- Github: gatorsmile

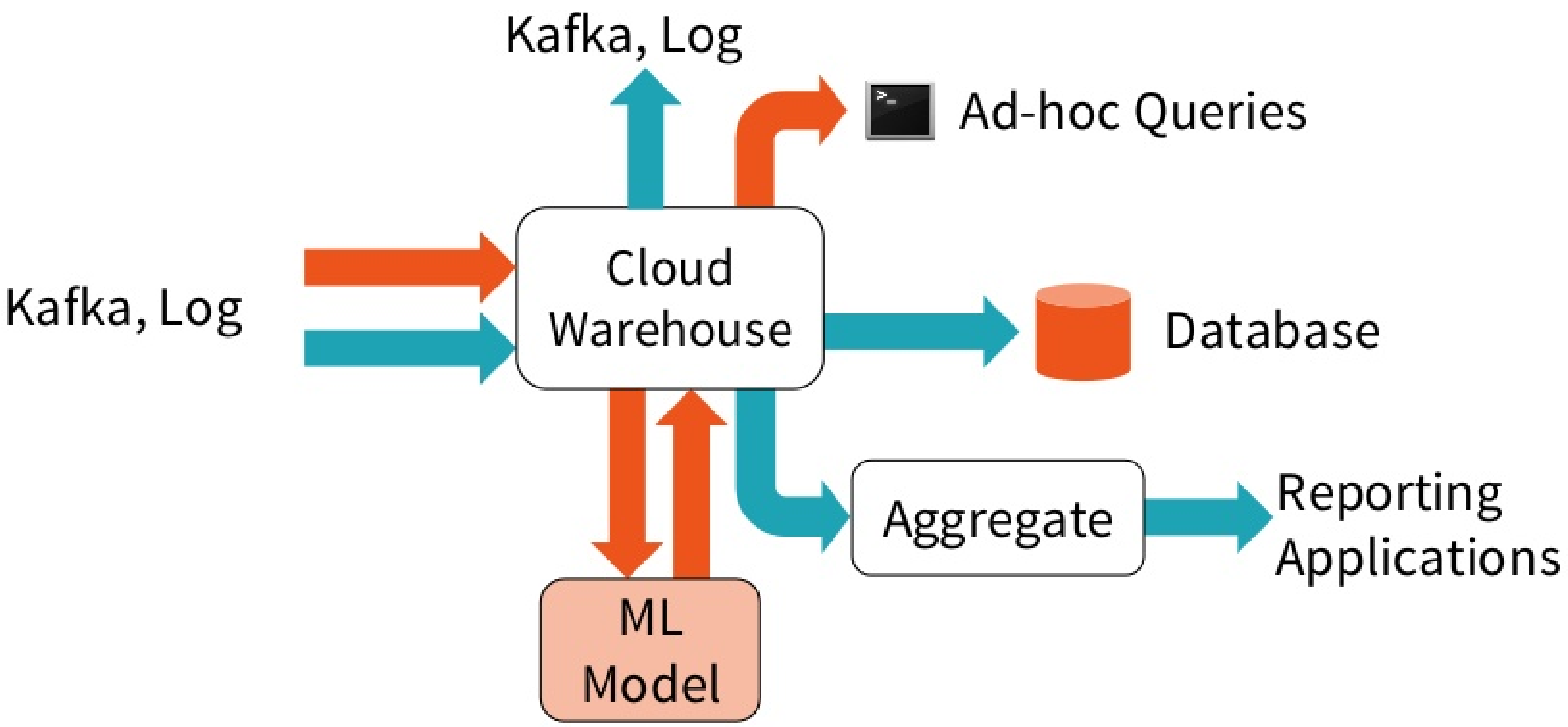
Overview

1. What's an ETL Pipeline?
2. Using Spark SQL for ETL
   - Extract: Dealing with Dirty Data (Bad Records or Files)
   - Extract: Multi-line JSON/CSV Support
   - Transformation: High-order functions in SQL
   - Load: Unified write paths and interfaces
3. New Features in Spark 2.3
   - Performance (Data Source API v2, Python UDF)

What is a Data Pipeline?

1. Sequence of transformations on data
2. Source data is typically semi-structured/unstructured (JSON, CSV etc.) and structured (JDBC, Parquet, ORC, the other Hive-serde tables)
3. Output data is integrated, structured and curated.
   - Ready for further data processing, analysis and reporting

Example of a Data Pipeline



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 sources (EXTRACT)
   - Transform data into a consumable format (TRANSFORM)
   - Transmit data to downstream consumers (LOAD)

An ETL Query in Apache Spark

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

EXTRACT

TRANSFORM

LOAD

An ETL Query in Apache Spark

```
val csvTable = spark.read.csv("/source/path")
val jdbcTable = spark.read.format("jdbc")
  .option("url", "jdbc:postgresql:...")
  .option("dbtable", "TEST.PEOPLE")
  .load()
csvTable
  .join(jdbcTable, Seq("name"), "outer")
  .filter("id <= 2999")
  .write
  .mode("overwrite")
  .format("parquet")
  .saveAsTable("outputTableName")
```

EXTRACT

TRANSFORM

LOAD

What's so hard about ETL Queries?

Why is ETL Hard?

1. Various sources/formats
2. Schema mismatch
3. Different representation
4. Corrupted files and data
5. Scalability
6. Schema evolution
7. Continuous ETL

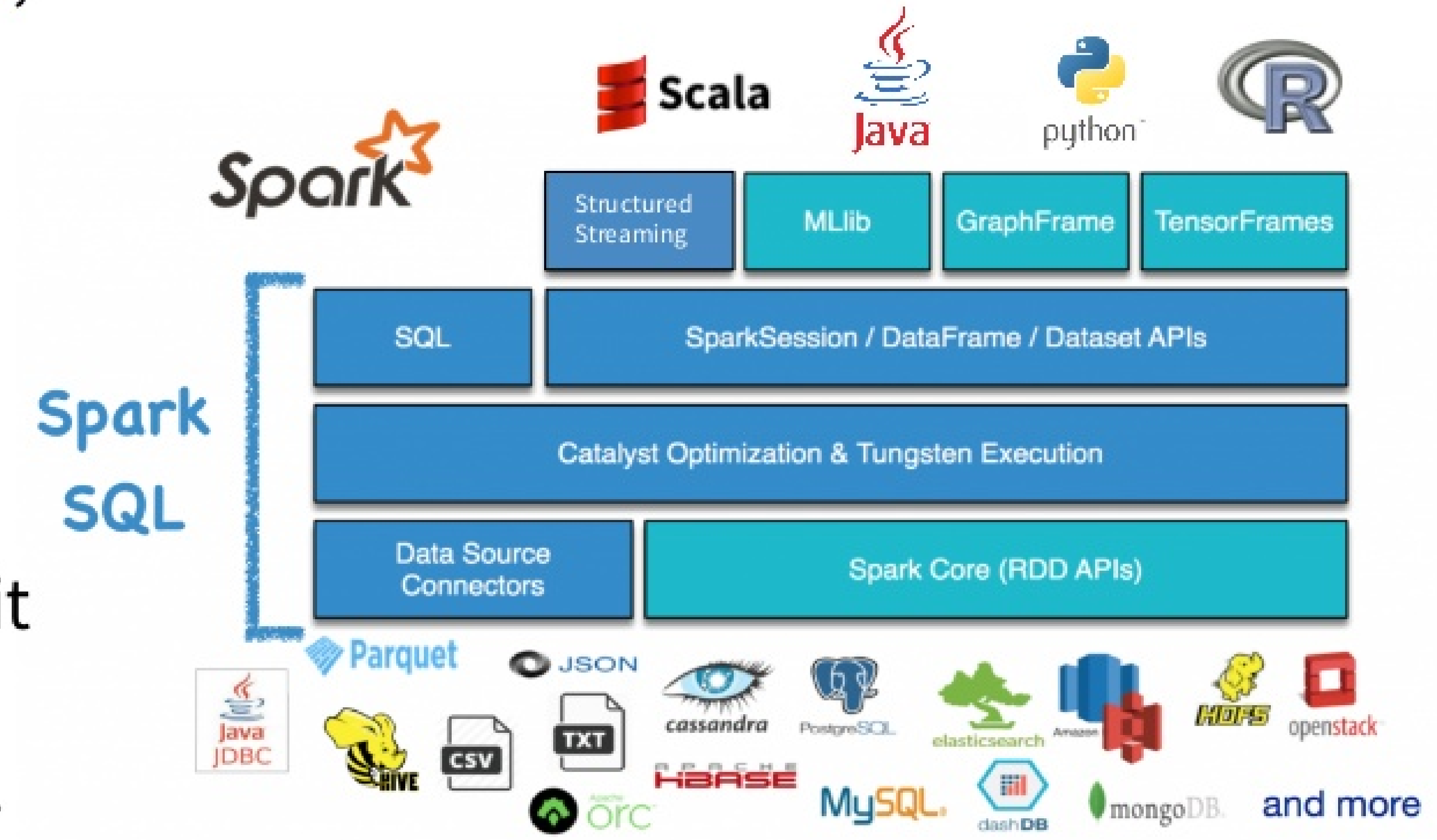
1. Too complex
2. Error-prone
3. Too slow
4. Too expensive

This is why **ETL** is **important**

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

Using Spark SQL for ETL

Spark SQL's flexible APIs, support for a wide variety of datasources, build-in support for structured streaming, state of art catalyst optimizer and tungsten execution engine make it a great framework for building end-to-end ETL pipelines.



Data Source Supports

1. Built-in connectors in Spark:
   - JSON, CSV, Text, Hive, Parquet, ORC, JDBC

2. Third-party data source connectors:
   - https://spark-packages.org

3. Define your own data source connectors by Data Source APIs
   - Ref link: https://youtu.be/uxuLRiNoDio

Schema Inference – semi-structured files

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

```
spark.read
 .json("/source/path")
 .printSchema()
```

```
root
 |-- a: long (nullable = true)
 |-- b: long (nullable = true)
 |-- c: long (nullable = true)
 |-- d: long (nullable = true)
 |-- e: long (nullable = true)
```

Schema Inference – semi-structured files

```
{"a":1, "b":2, "c":3.1}
{"e":2, "c":3, "b":5}
{"a":"5", "d":7}
```

```
spark.read
 .json("/source/path")
 .printSchema()
```

```
root
 |-- a: string (nullable = true)
 |-- b: long (nullable = true)
 |-- c: double (nullable = true)
 |-- d: long (nullable = true)
 |-- e: long (nullable = true)
```

User-specified Schema

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

```
+----+----+
|   a|   b|
+----+----+
|   1|   2|
|null|   5|
|   5|null|
+----+----+
```

```
val schema = new StructType()
  .add("a", "int")
  .add("b", "int")

spark.read
  .json("/source/path")
  .schema(schema)
  .show()
```

User-specified DDL-format Schema

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

```
+----+----+
|   a|   b|
+----+----+
|   1|   2|
|null|   5|
|   5|null|
+----+----+
```

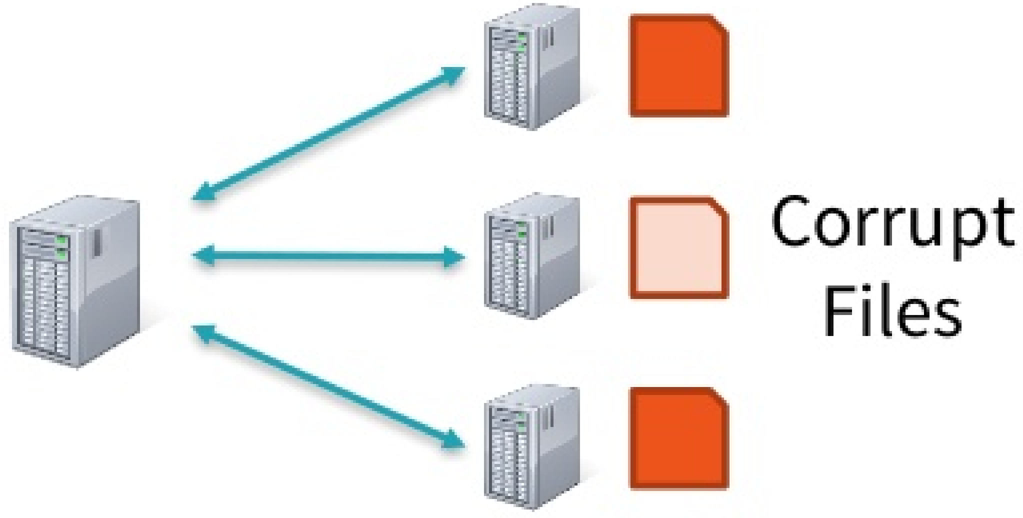
```
spark.read
  .json("/source/path")
  .schema("a INT, b INT")
  .show()
```

Availability: Apache Spark 2.2

Dealing with Bad Data: Skip Corrupt Files

java.io.IOException. For example, java.io.EOFException: Unexpected end of input stream at org.apache.hadoop.io.compress.DecompressorStream.decompress

java.lang.RuntimeException: file:/temp/path/c000.json is not a Parquet file (too small)



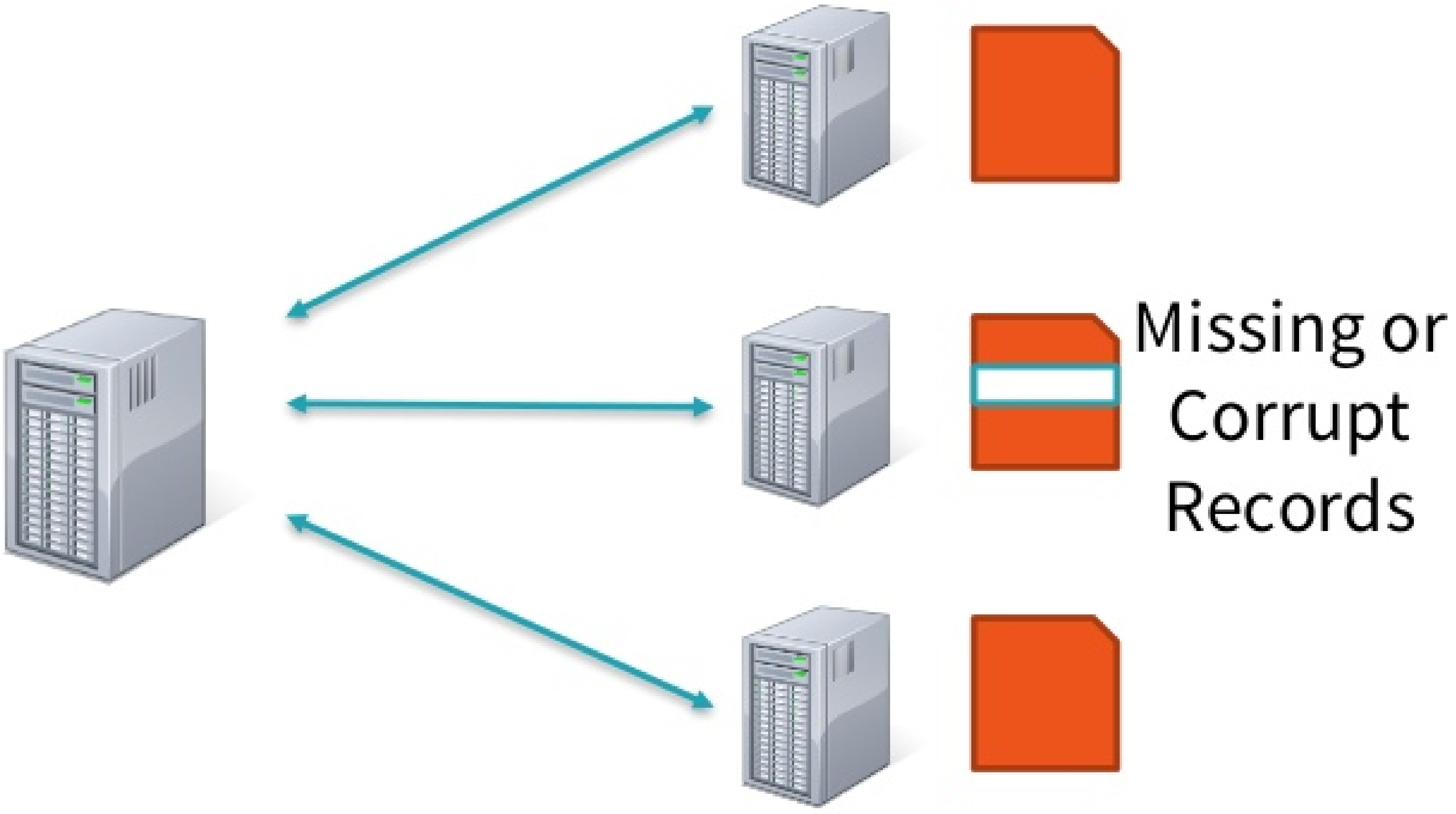
[SPARK-17850] If true, the Spark jobs will continue to run even when it encounters corrupt files. The contents that have been read will still be returned.

spark.sql.files.ignoreCorruptFiles = true

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}
```

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

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

The default can be configured via spark.sql.columnNameOfCorruptRecord

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}
```

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
  .option("mode", "FAILFAST")
  .csv(corruptRecords)
  .show()
```

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

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.
  .option("mode", "PERMISSIVE")
  .csv(corruptRecords)
  .show()
```

```
+----+-------+-----+--------------------+-----+
| _c0|    _c1|  _c2|                 _c3|  _c4|
+----+-------+-----+--------------------+-----+
|year|   make|model|             comment|blank|
|2012|"Tesla"|  "S"|        "No comment"| null|
|1997|   Ford| E350|"Go get one now ...| 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
  .option("header", true)
  .option("mode", "PERMISSIVE")
  .csv(corruptRecords)
  .show()
```

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

CSV: Dealing with Corrupt Records

```
val schema = "col1 INT, col2 STRING, col3 STRING, col4 STRING, " +
  "col5 STRING, __corrupted_column_name STRING"
spark.read
  .option("header", true)
  .option("mode", "PERMISSIVE")
  .csv(corruptRecords)
  .show()
```

```
+----+--------+-----+--------------------+----+-----------------------+
|col1|    col2| col3|                col4|col5|__corrupted_column_name|
+----+--------+-----+--------------------+----+-----------------------+
|2012| "Tesla"|  "S"|        "No comment"|null|                   null|
|1997|    Ford| E350|"Go get one now ...|null|                   null|
|2015|   Chevy| Volt|                null|null|      2015, Chevy, Volt|
+----+--------+-----+--------------------+----+-----------------------+
```

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
  .option("mode", "DROPMALFORMED")
  .csv(corruptRecords)
  .show()
```

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

Functionality: Better Corruption Handling

badRecordsPath: a user-specified path to store exception files for recording the information about bad records/files.

- A unified interface for both corrupt records and files
- Enabling multi-phase data cleaning
- DROPMALFORMED + Exception files
  - No need an extra column for corrupt records
  - Recording the exception data, reasons and time.

Availability: Databricks Runtime 3.0

Functionality: Better JSON and CSV Support

[SPARK-18352] [SPARK-19610] **Multi-line** JSON and CSV Support

- Spark SQL currently reads JSON/CSV one line at a time
- Before 2.2, it requires custom ETL

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

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

Availability: Apache Spark 2.2

Transformation: Higher-order Function in SQL

Transformation on complex objects like arrays, maps and structures inside of columns.

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

UDF ? Expensive data serialization

Transformation: Higher order function in SQL

Transformation on complex objects like arrays, maps and structures inside of columns.

1) Check for element existence

```
SELECT EXISTS(values, e -> e > 30) AS v
FROM tbl_nested;
```

2) Transform an array

```
SELECT TRANSFORM(values, e -> e * e) AS v
FROM tbl_nested;
```

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

Transformation: Higher order function in SQL

3) Filter an array

```
SELECT FILTER(values, e -> e > 30) AS v
FROM tbl_nested;
```

4) Aggregate an array

```
SELECT REDUCE(values, 0, (value, acc) -> value + acc) AS sum
FROM tbl_nested;
```

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

Ref Databricks Blog: http://dbricks.co/2rUKQ1A

More cool features available in DB Runtime 3.0: http://dbricks.co/2rhPM4c

Availability: Databricks Runtime 3.0

New Format in DataframeWriter API

Users can create Hive-serde tables using DataframeWriter APIs

```
df.write.format("hive")
  .option("fileFormat", "avro")
  .saveAsTable("tab")
```

CREATE Hive-serde tables

```
df.write.format("parquet")
  .saveAsTable("tab")
```

CREATE data source tables

Availability: Apache Spark 2.2

Unified CREATE TABLE [AS SELECT]

```
CREATE TABLE t1(a INT, b INT)
STORED AS ORC
```

```
CREATE TABLE t1(a INT, b INT)
USING hive
OPTIONS(fileFormat 'ORC')
```

CREATE Hive-serde tables

```
CREATE TABLE t1(a INT, b INT)
USING ORC
```

CREATE data source tables

Availability: Apache Spark 2.2

Unified CREATE TABLE [AS SELECT]

Apache Spark preferred syntax

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS]
  [db_name.]table_name
USING table_provider
[OPTIONS table_property_list]
[PARTITIONED BY (col_name, col_name, ...)]
[CLUSTERED BY (col_name, col_name, ...)
  [SORTED BY (col_name [ASC|DESC], ...)]
  INTO num_buckets BUCKETS]
[LOCATION path]
[COMMENT table_comment]
[AS select_statement];
```

Availability: Apache Spark 2.2

Apache Spark 2.3+

Massive focus on building ETL-friendly pipelines

[SPARK-15689] Data Source API v2

1. [SPARK-20960] An efficient column batch interface for data exchanges between Spark and external systems.
   - Cost for conversion to and from RDD[Row]
   - Cost for serialization/deserialization
   - Publish the columnar binary formats
2. Filter pushdown and column pruning
3. Additional pushdown: limit, sampling and so on.

Target: Apache Spark 2.3

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. Improve data exchange between Python and JVM
5. Block-level UDFs
   - Block-level arguments and return types

Target: Apache Spark 2.3

Recap

1. What's an ETL Pipeline?
2. Using Spark SQL for ETL
   - Extract: Dealing with Dirty Data (Bad Records or Files)
   - Extract: Multi-line JSON/CSV Support
   - Transformation: High-order functions in SQL
   - Load: Unified write paths and interfaces
3. New Features in Spark 2.3
   - Performance (Data Source API v2, Python UDF)

Try Apache Spark in Databricks!

UNIFIED ANALYTICS PLATFORM

- Collaborative cloud environment
- Free version (community edition)

DATABRICKS RUNTIME 3.0

- Apache Spark - optimized for the cloud
- Caching and optimization layer - DBIO
- Enterprise security - DBES

Try for free today.
databricks.com

Questions?

Xiao Li (lixiao@databricks.com)