



Lessons From the Field: Applying Best Practices to Your Apache Spark™ Applications

Silvio Fiorito

Spark Summit Europe, 2017

#EUdev5



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

- Silvio Fiorito
 - silvio@databricks.com
 - [@granturing](https://twitter.com/granturing)
- Resident Solutions Architect @ Databricks
- Spark developer, trainer, consultant - using Spark since ~v0.6
- Prior to that - application security, cyber analytics, forensics, etc.



Outline

- Speeding Up File Loading & Partition Discovery
- Optimizing File Storage & Layout
- Identifying Bottlenecks In Your Queries

“Why is it taking so long to
‘load’ my DataFrame”

“Loading” DataFrames

```
Cmd 1
```

```
1 val df1 = spark.read.load("/tmp/tpcds/store_sales")
2   .filter('ss_sold_date_sk isin (2450816,2450835,2450845,2450860))
```

▼ (2) Spark Jobs

- ▼ Job 0 [View](#) (Stages: 1/1)
Stage 0: 1823/1823 ⓘ
- ▶ Job 1 [View](#) (Stages: 1/1)

df1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [ss_sold_time_s
k: int, ss_item_sk: int ... 21 more fields]

Command took 42.66 seconds -- by silvio@databricks.com at 9/5/2017, 2:00:55 PM on silviotest

“Loading” DataFrames

Spark is lazily executed, right?



The screenshot shows a Databricks command window titled "Cmd 1". It contains two lines of Scala code: `1 val df1 = spark.read.load("/tmp/tpcds/store_sales")` and `2 .filter('ss_sold_date_sk isin (2450816,2450835,2450845,2450860))`. Below the code, there is a section for Spark Jobs showing two jobs, Job 0 and Job 1, both with 1 stage. Job 0's stage 0 is 1823/1823. Below the jobs, the output shows the DataFrame `df1` as a `org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]` with fields `ss_sold_time_s`, `k: int`, `ss_item_sk: int`, and 21 more fields. At the bottom, it states the command took 42.66 seconds and was executed by `silvio@databricks.com` on 9/5/2017 at 2:00:55 PM on `silviotest`.

```
Cmd 1
```

```
1 val df1 = spark.read.load("/tmp/tpcds/store_sales")
2   .filter('ss_sold_date_sk isin (2450816,2450835,2450845,2450860))
```

▼ (2) Spark Jobs


- ▼ Job 0 [View](#) (Stages: 1/1)
Stage 0: 1823/1823 ⓘ
- ▶ Job 1 [View](#) (Stages: 1/1)

df1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [ss_sold_time_s
k: int, ss_item_sk: int ... 21 more fields]

Command took 42.66 seconds -- by silvio@databricks.com at 9/5/2017, 2:00:55 PM on silviotest

“Loading” DataFrames

Spark is lazily executed, right?



The screenshot shows a Databricks command window titled "Cmd 1". It contains two lines of Scala code: `1 val df1 = spark.read.load("/tmp/tpcds/store_sales")` and `2 .filter('ss_sold_date_sk isin (2450816,2450835,2450845,2450860))`. Below the code, the "Spark Jobs" section is expanded, showing two jobs: "Job 0" and "Job 1", both with "Stages: 1/1". Job 0's stage is "Stage 0: 1823/1823". Below the jobs, the output shows the DataFrame type: `df1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]` and its schema: `ss_sold_date_sk: int, ss_item_sk: int ... 21 more fields]`. At the bottom, it states "Command took 42.66 seconds -- by silvio@databricks.com at 9/5/2017, 2:00:55 PM on silviotest".

```
Cmd 1
```

```
1 val df1 = spark.read.load("/tmp/tpcds/store_sales")
2   .filter('ss_sold_date_sk isin (2450816,2450835,2450845,2450860))
```

▼ (2) Spark Jobs

- ▼ Job 0 [View](#) (Stages: 1/1)
Stage 0: 1823/1823 ⓘ
- ▶ Job 1 [View](#) (Stages: 1/1)

df1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]
ss_sold_date_sk: int, ss_item_sk: int ... 21 more fields]

Command took 42.66 seconds -- by silvio@databricks.com at 9/5/2017, 2:00:55 PM on silviotest

Jobs running even though there's no action, why?

Datasource API

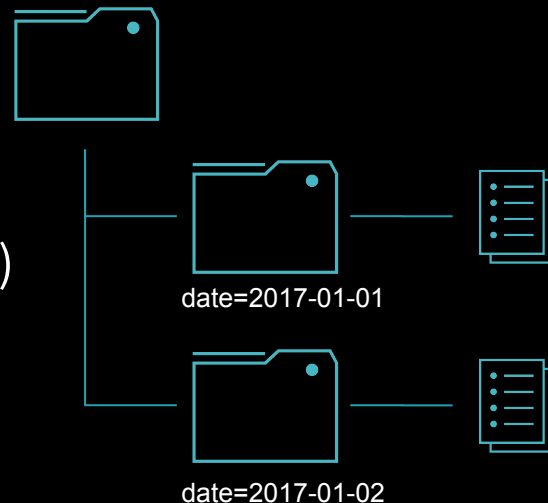
Datasource

- Maps “spark.read.load” command to underlying data source (Parquet, CSV, ORC, JSON, etc.)
- For file sources - Infers schema from files
 - Kicks off Spark job to parallelize
 - Needs file listing first
- Need basic statistics for query planning (file size, partitions, etc.)

Listing Files

InMemoryFileIndex

- Discovers partitions & lists files, using Spark job if needed
([spark.sql.sources.parallelPartitionDiscovery.threshold](#) 32 default)
- [FileStatusCache](#) to cache file status ([250MB default](#))
- Maps Hive-style partition paths into columns
- Handles partition pruning based on query filters



Partition Discovery

```
Cmd 1
```

```
1 val df1 = spark.read.load("/tmp/tpcds/store_sales")
2   .filter('ss_sold_date_sk isin (2450816,2450835,2450845,2450860))
```

▼ (2) Spark Jobs

- ▼ Job 0 [View](#) (Stages: 1/1)
Stage 0: 1823/1823 ⓘ
- ▶ Job 1 [View](#) (Stages: 1/1)

df1: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row]
k: int, ss_item_sk: int ... 21 more fields]

Command took 42.66 seconds -- by silvio@databricks.com at 9/5/2017, 2:00:55 PM on silviotest

1,823 partition paths to index

Partition Pruning

Cmd 2

```
1 df1.explain|
```

== Physical Plan ==

```
*FileScan parquet [ss_sold_time_sk#7,ss_item_sk#8,ss_customer_sk#9,ss_demo_sk#10,ss_hdemo_sk#11,ss_addr_sk#12,ss_store_sk#13,ss_promo_sk#14,ss_quantity#15L,ss_quantity#16,ss_wholesale_cost#17,ss_list_price#18,ss_sales_price#19,ss_ext_discount_amt#20,ss_ext_sales_price#21,ss_ext_wholesale_cost#22,ss_ext_list_price#23,ss_ext_tax#24,ss_coupon_amt#25,ss_net_paid#26,ss_net_paid_inc_tax#27,ss_net_profit#28,ss_sold_date_sk#29] Batched: true, Format: Parquet, Location: InMemoryFileIndex[dbfs:/tmp/tpcds/store_sales], PartitionCount: 4, PartitionFilters: [ss_sold_date_sk#29 IN (2450816,2450835,2450845,2450860)], PushedFilters: [], ReadSchema: struct<ss_sold_time_sk:int,ss_item_sk:int,ss_customer_sk:int,ss_demo_sk:int,ss_hdemo_sk:int,ss_addr_sk:int,ss_store_sk:int,ss_promo_sk:int,ss_quantity:int,ss_wholesale_cost:double,ss_list_price:double,ss_sales_price:double,ss_ext_discount_amt:double,ss_ext_sales_price:double,ss_ext_wholesale_cost:double,ss_ext_list_price:double,ss_ext_tax:double,ss_coupon_amt:double,ss_net_paid:double,ss_net_paid_inc_tax:double,ss_net_profit:double,ss_sold_date_sk:int>
```

Command took 0.45 seconds -- by silvio@databricks.com at 9/5/2017, 2:14:50 PM on silviotest

We only care about 4 out of 1,823 partitions!!

Dealing With Many Partitions

Option 1:

- If you know exactly what partitions you want
- If you need to use files vs Datasource Tables
- Specify “basePath” option and load specific partition paths
- InMemoryFileIndex “pre-filters” on those paths

Dealing With Many Partitions

```
Cmd 4

1  val df2 = spark.read.option("basePath", "/tmp/tpcds/store_sales")
2    .load(
3      "/tmp/tpcds/store_sales/ss_sold_date_sk=2450816",
4      "/tmp/tpcds/store_sales/ss_sold_date_sk=2450835",
5      "/tmp/tpcds/store_sales/ss_sold_date_sk=2450845",
6      "/tmp/tpcds/store_sales/ss_sold_date_sk=2450860"
7    )

▶ (1) Spark Jobs

df2: org.apache.spark.sql.DataFrame = [ss_sold_time_sk: int, ss_item_sk: int
... 21 more fields]

Command took 2.10 seconds -- by silvio@databricks.com at 9/5/2017, 2:14:10 PM on silviotest
```

Dealing With Many Partitions

```
Cmd 4

1  val df2 = spark.read.option("basePath", "/tmp/tpcds/store_sales")
2    .load(
3      "/tmp/tpcds/store_sales/ss_sold_date_sk=2450816",
4      "/tmp/tpcds/store_sales/ss_sold_date_sk=2450835",
5      "/tmp/tpcds/store_sales/ss_sold_date_sk=2450845",
6      "/tmp/tpcds/store_sales/ss_sold_date_sk=2450860"
7    )

spark.sql.DataFrame = [ss_sold_time_sk: int, ss_item_sk: int
... 2 more fields]

Command took 2.10 seconds -- by silvio@databricks.com at 9/5/2017, 2:14:10 PM on silviotest
```

~2 seconds vs ~43 seconds

Dealing With Many Partitions

Option 2:

- Use Datasource Tables with Spark 2.1+
- Partitions managed in Hive metastore
- Partition Pruning at logical planning stage
- [Scalable Partition Handling for Cloud-Native Architecture in Apache 2.1](#)

Dealing With Many Partitions

```
Cmd 7

1 val df3 = spark.read.table("tpcds.store_sales")
2   .filter('ss_sold_date_sk isin (2450816,2450835,2450845,2450860))

df3: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [ss_sold_time_s
k: int, ss_item_sk: int ... 21 more fields]

Command took 0.29 seconds -- by silvio@databricks.com at 9/5/2017, 2:28:55 PM on silviotest
```

~0.29 seconds vs ~43
seconds

Using Datasource Tables

Using SQL - External (or Unmanaged) Tables

- Create over existing data
- Partition discovery runs once
- Use “saveAsTable” or “insertInto” to add new partitions

Cmd 3

```
1 CREATE TABLE IF NOT EXISTS tpcds_temp.store_sales
2 USING parquet
3 OPTIONS (path '/tmp/tpcds/store_sales');
4
5 MSCK REPAIR TABLE tpcds_temp.store_sales;
```

Using Datasource Tables

Using SQL - Managed Tables

- SparkSQL manages metadata and underlying files
- Use “saveAsTable” or “insertInto” to add new partitions

```
Cmd 2

1 CREATE TABLE IF NOT EXISTS tpcds_temp.store_sales (
2     ss_sold_time_sk int,
3     ss_item_sk int,
4     ss_customer_sk int,
5     ss_sold_date_sk int
6 )
7 USING parquet
8 PARTITIONED BY (ss_sold_date_sk);
```

Using Datasource Tables

Using DataFrame API

- Use “saveAsTable” or “insertInto” to add new partitions

```
Cmd 10
1 df1.write
2   .partitionBy("ss_sold_date_sk")
3   .saveAsTable("tpcds_temp|store_sales")
```

Managed Table

Using Datasource Tables

Using DataFrame API

- Use “saveAsTable” or “insertInto” to add new partitions

```
Cmd 10
1 df1.write
2   .partitionBy("ss_sold_date_sk")
3   .saveAsTable("tpcds_temp.store_sales")
```

Managed Table

```
Cmd 10
1 df1.write
2   .partitionBy("cs_sold_date_sk")
3   .option("path", "/tmp/tpcds_temp/stores_sales")
4   .saveAsTable("tpcds_temp.store_sales")
```

Unmanaged Table

Datasource Tables vs Files

TABLES

- Managed, more scalable partition handling
- Schema in metastore
- Faster job startup
- Additional statistics available to Catalyst (e.g. CBO)
- Use SQL or DataFrame API

FILES

- Partition discovery at each DataFrame creation
- Infer schema from files
- Slower job startup time
- Only file-size statistics available
- Only DataFrame API (SQL with temp views)

Reading CSV & JSON Files

Schema Inference

- Runs on whole dataset
- Convenient, but SLOW... delays job startup
- Even worse with poor file layout
 - Large GZIP files
 - Lots of small files

Reading CSV & JSON Files

Avoiding Schema Inference

- Easiest - use Tables!
- Otherwise...If consistent data, infer on subset of files
- Save schema for reuse (e.g. scheduled batch jobs)
- JSON - adjust [samplingRatio](#) (default 1.0)

Better Schema Inference

Cmd 2

```
1 val df = spark.read.format("csv")
2   .option("header", true)
3   .option("inferSchema", true)
4   .load("/tmp/tpcds_csv/store_sales")
```

▶ (3) Spark Jobs

df: org.apache.spark.sql.DataFrame = [ss_sold_time_sk: int, ss_item_sk: int ... 21 more fields]

Command took 48.10 seconds -- by silvio@databricks.com at 9/15/2017, 3:31:39 PM on silviotest

Better Schema Inference

```
Cmd 4

1  val schemaDF = spark.read.format("csv")
2    .option("header", true)
3    .option("inferSchema", true)
4    .load(sampleFile)
5
6  val df = spark.read.format("csv")
7    .option("header", true)
8    .schema(schemaDF.schema)
9    .load("/tmp/tpcds_csv/store_sales")

▶ (2) Spark Jobs

schemaDF: org.apache.spark.sql.DataFrame = [ss_sold_time_sk: int, ss_item_sk: int
... 21 more fields]
df: org.apache.spark.sql.DataFrame = [ss_sold_time_sk: int, ss_item_sk: int ... 21 m
ore fields]

Command took 6.60 seconds - by silvio@databricks.com at 9/15/2017, 3:42:11 PM on silviotest
```

Better Schema Inference

Cmd 6

```
1 import org.apache.spark.sql.types._
2
3 lazy val useSchema = DataType.fromJson(schemaJSON)
4   .asInstanceOf[StructType]
5
6 val df = spark.read.format("csv")
7   .option("header", true)
8   .schema(useSchema)
9   .load("/tmp/tpcds_csv/store_sales")
```

```
import org.apache.spark.sql.types._
useSchema: org.apache.spark.sql.types.StructType = <lazy>
df: org.apache.spark.sql.DataFrame = [ss_sold_time_sk: int, ss_item_sk: int ... 21 more fields]
```

Command took 0.59 seconds -- by silvio@databricks.com at 9/15/2017, 3:48:39 PM on silviotest

“What format, compression,
and partitioning scheme
should I use?”

Managing & Optimizing File Output

File Types

- Prefer columnar over text for analytical queries
- Parquet
 - Column pruning
 - Predicate pushdown
- CSV/JSON
 - Must parse whole row
 - No column pruning
 - No predicate pushdown

Managing & Optimizing File Output

Compression

- Prefer splittable
 - LZ4, BZip2, LZO, etc.
- Parquet + Snappy or GZIP (splittable due to row groups)
 - Snappy is default in Spark 2.2+
- **AVOID** - Large GZIP text files
 - Not splittable
 - GC issues with wide tables
- More - [Why You Should Care about Data Layout in the Filesystem](#)

Managing & Optimizing File Output

PARTITIONING

- Coarse-grained filtering of input files
- Avoid over-partitioning (small files), overly nested partitions

```
output.write
  .partitionBy("ss_sold_date_sk")
  .option("path", "/tmp/tpcds_temp/store_sales_temp")
  .saveAsTable("tpcds_temp.store_sales_temp")
```

BUCKETING

- Write data already hash-partitioned
- Good for joins or aggregations (avoids shuffle)
- Must be Datasource table

```
output.write
  .bucketBy(100, "ss_item_sk")
  .sortBy("ss_item_sk")
  .option("path", "/tmp/tpcds_temp/store_sales_temp")
  .saveAsTable("tpcds_temp.store_sales_temp")
```

Partitioning & Bucketing

Writing Partitioned & Bucketed Data

- Each task writes a file per-partition and bucket
- If data is randomly distributed across tasks...lots of small files!
- Coalesce?
 - Doesn't guarantee colocation of partition/bucket values
 - Reduces parallelization of final stage
- Repartition - incurs a shuffle but results in cleaner files
- Compaction - run job periodically to generate clean files

Managing File Sizes

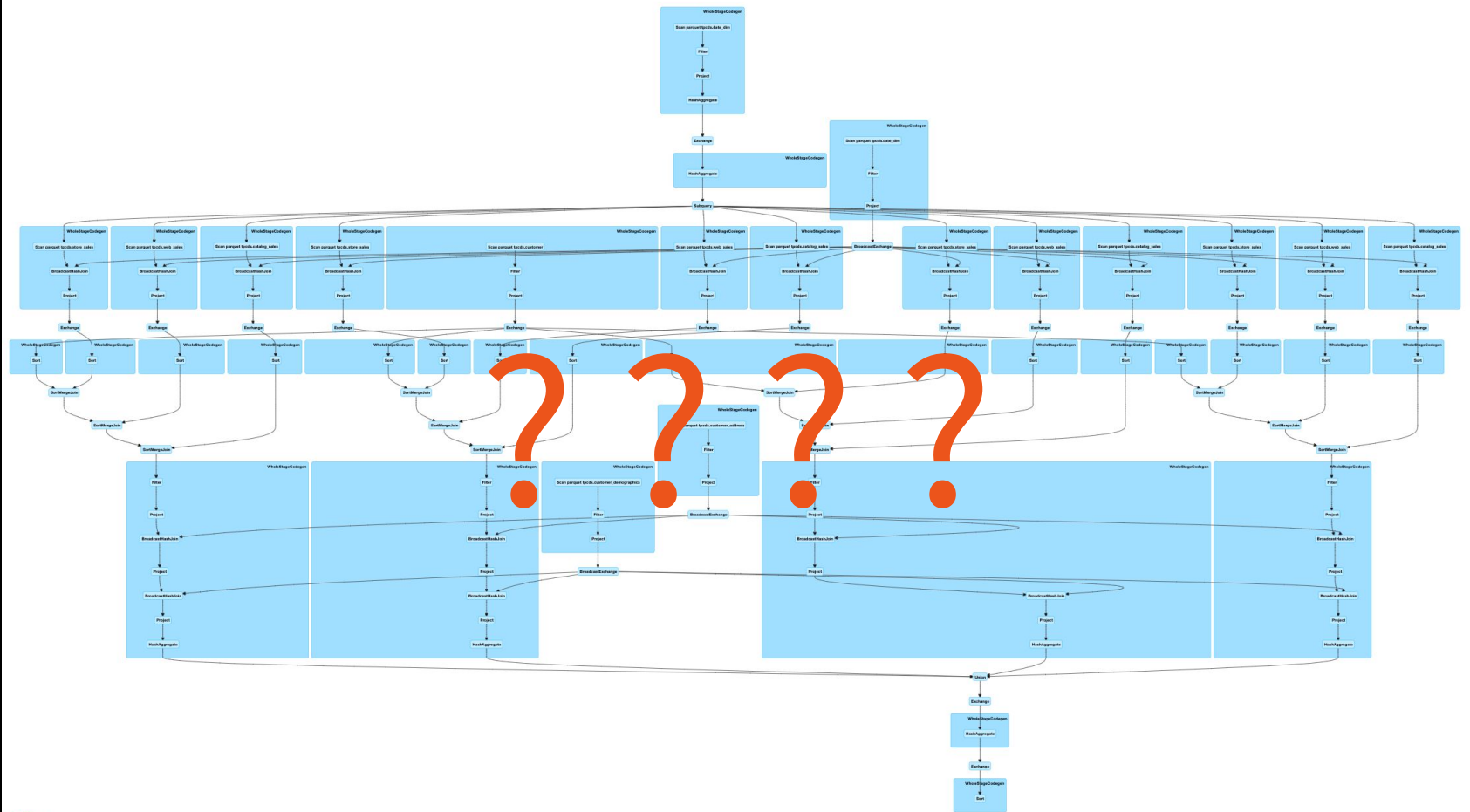
```
output.repartition('ss_sold_date_sk').write  
  .partitionBy("ss_sold_date_sk")  
  .option("path", "/tmp/tpcds_temp/store_sales_temp")  
  .saveAsTable("tpcds_temp.store_sales_temp")
```

Managing File Sizes

What If Partitions Are Too Big

- Spark 2.2 - [spark.sql.files.maxRecordsPerFile](#) (default disabled)
- Write task is responsible for splitting records across files
 - WARNING: not parallelized
- If need parallelization - repartition with additional key
 - Use hash+mod to manage # of files

“How can I optimize my
query?”



Details

Managing Shuffle Partitions

Spark SQL Shuffle Partitions

- Default to 200, used in shuffle operations
 - groupBy, repartition, join, window
- Depending on your data volume might be too small/large
- Override with conf - spark.sql.shuffle.partitions
 - 1 value for whole query

Managing Shuffle Partitions

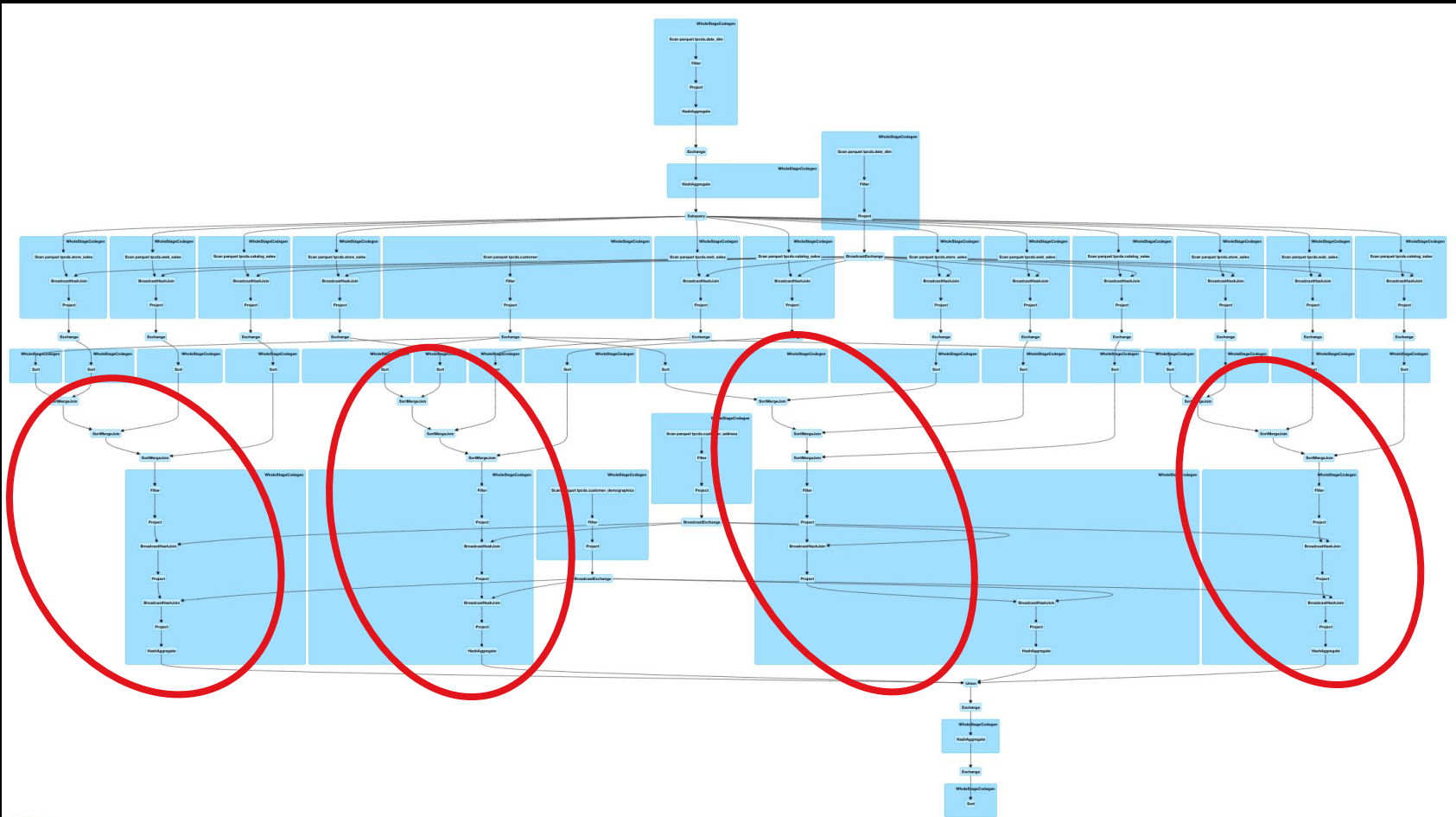
Adaptive Execution

- Introduced in Spark 2.0
- Sets shuffle partitions based on size of shuffle output
- [spark.sql.adaptive.enabled](#) (default false)
- [spark.sql.adaptive.shuffle.targetPostShuffleInputSize](#) (default 64MB)
- Still in development* - [SPARK-9850](#)

Unions

Understanding unions

- Each DataFrame in union runs independently until shuffle
- Self-union - DataFrame read N times (number of unions)
- Alternatives (depends on use case)
 - explode or flatMap
 - persist root DataFrame (read once from storage)



Details

Optimizing Query Execution

Cost Based Optimizer

- Added in Spark 2.2
- Collects and uses per-column stats for query planning
- Requires Datasource Tables
- [Cost Based Optimizer in Apache Spark 2.2](#)

Computing Statistics for CBO

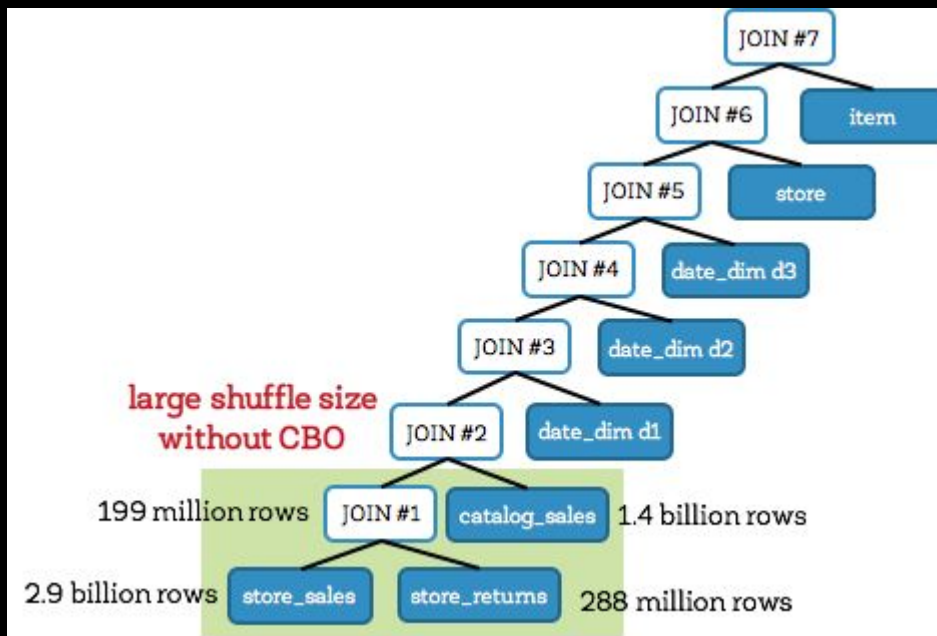
Cmd 9

```
1 --compute table-level statistics (# of rows, size)
2 ANALYZE TABLE tpcds.store_sales COMPUTE STATISTICS
```

Cmd 10

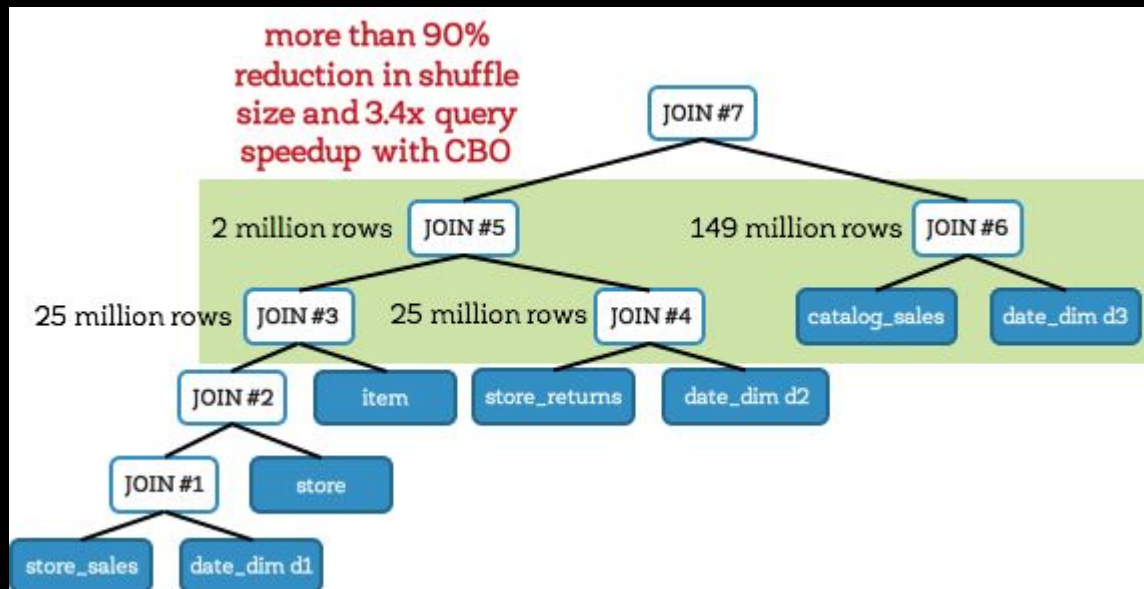
```
1 --compute column-level statistics
2 ANALYZE TABLE tpcds.store_sales COMPUTE STATISTICS
3 FOR COLUMNS ss_sales_price, ss_item_sk, ss_quantity
```

TPC-DS Query 25 Without CBO



<https://databricks.com/blog/2017/08/31/cost-based-optimizer-in-apache-spark-2-2.html>

TPC-DS Query 25 With CBO



<https://databricks.com/blog/2017/08/31/cost-based-optimizer-in-apache-spark-2-2.html>



Data Skipping Index

- Added in Databricks Runtime 3.0
- Use file-level stats to skip files based on filters
- Requires Datasource Tables
- Opt-in, no code changes
- [Data Skipping Index Docs](#)

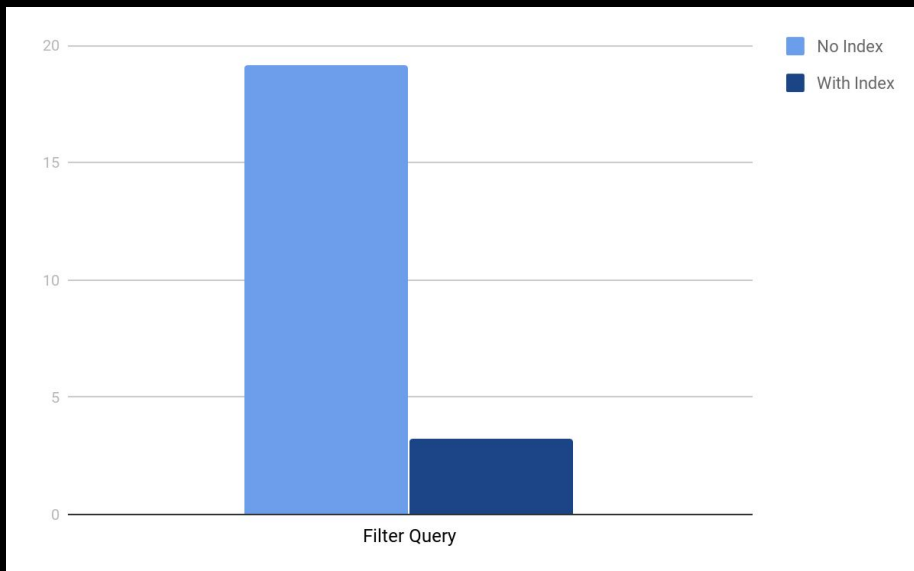
```
Cmd 11  
1 CREATE DATASKIPPING INDEX ON TABLE tpcds.store_sales
```



Data Skipping Index

Cmd 16

```
1 df1.filter('ss_customer_sk between (1, 10)).foreach { _ => }
```



Try Apache Spark in Databricks!

UNIFIED ANALYTICS PLATFORM

- Collaborative cloud environment
- Free version (community edition)

DATABRICKS RUNTIME 3.3

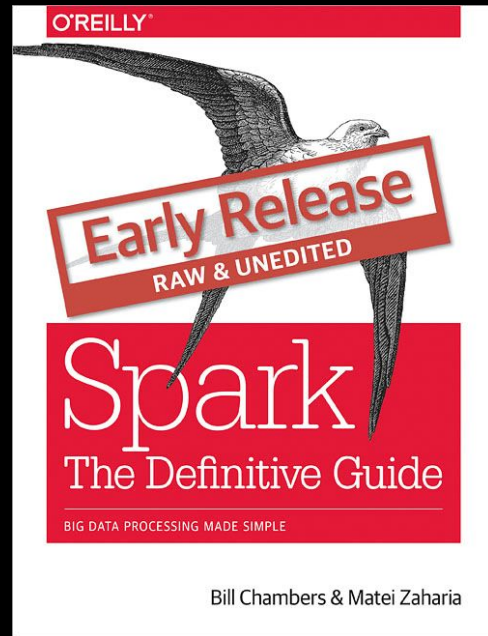
- Apache Spark - optimized for the cloud
- Caching and optimization layer - DBIO
- Enterprise security - DBES

Try for free today.
databricks.com

Spark the Definitive Guide

Early Release

- <http://go.databricks.com/definitive-guide-apache-spark>
- Blog Post on [Preview of Apache Spark: The Definitive Guide](#)





Thank you

Q&A

silvio@databricks.com
[@granturing](#)