# Keeping Spark on Track:
Productionizing Spark for ETL

Kyle Pistor
kyle@databricks.com
Miklos Christine
mwc@databricks.com

## $ whoami

Kyle Pistor

- SA @ Databricks
- 100s of Customers
- Focus on ETL and big data warehousing using Apache Spark
- BS/MS - EE

Miklos Christine

- SA @ Databricks!
- Previously: Systems Engineer @ Cloudera
- Deep Knowledge of Big Data Stack
- Apache Spark Enthusiast

databricks

# Agenda

1. **ETL:** Why ETL?
2. **Schemas**
3. **Metadata:** Best Practices FS Metadata
4. **Performance:** Tips & Tricks
5. **Error Handling**

databricks

# Agenda

1. **ETL:** Why ETL?
2. **Schemas**
3. **Metadata:** Best Practices FS Metadata
4. **Performance:** Tips & Tricks
5. **Error Handling**

databricks

# Why ETL?

## Goal

- Transform raw files into more efficient binary format

## Benefits

- Performance improvements, statistics
- Space savings
- Standard APIs to different sources (CSV, JSON, etc.)
- More robust queries

## Agenda

1. **ETL:** Why ETL?
2. **Schemas**
3. **Metadata:** Best Practices FS Metadata
4. **Performance:** Tips & Tricks
5. **Error Handling**

databricks

## Schema Handling

### Common Raw Data

- Delimited (CSV, TSV, etc)
- JSON

### Infer Schema?

- Easy and quick

### Specify Schema

- Faster
- More robust

databricks

# JSON Record #1

```
{
    "time": 1486166400,
    "host": "my.app.com",
    "eventType": "test",
    "event": {
        "message": "something happened",
        "type": "INFO"
    }
}
```

databricks

## JSON Record #1

```
{
    "time": 1486166400,
    "host": "my.app.com",
    "eventType": "test",
    "event": {
        "message": "something happened",
        "type": "INFO"
    }
}
```

```
spark.read.json("record1.json").schema
```

```
root
 |-- event: struct (nullable = true)
 |    |-- message: string (nullable = true)
 |    |-- type: string (nullable = true)
 |-- host: string (nullable = true)
 |-- source: string (nullable = true)
 |-- time: long (nullable = true)
```

databricks

## JSON Record #2

```
{
    "time": 1486167800,
    "host": "my.app.com",
    "eventType": "test",
    "event": {
        "message": "Something else happened",
        "type": "INFO",
        "ip" : "59.16.12.0",
        "browser" : "chrome",
        "os" : "mac",
        "country" : "us"
    }
}
```

databricks

# JSON Record #2

```
{
    "time": 1486166400,
    "host": "my.app.com",
    "eventType": "test",
    "event": {
        "message": "something happened",
        "type": "INFO"
    }
}
```

```
{
    "time": 1486167800,
    "host": "my.app.com",
    "eventType": "test",
    "event": {
        "message": "Something else happened",
        "type": "INFO",
        "ip" : "59.16.12.0",
        "browser" : "chrome",
        "os" : "mac",
        "country" : "us"
    }
}
```

# JSON Record #1 & #2

```
{
    "time": 1486167800,
    "host": "my.app.com",
    "eventType": "test",
    "event": {
        "message": "Something else happened",
        "type": "INFO",
        "ip" : "59.16.12.0",
        "browser" : "chrome",
        "os" : "mac",
        "country" : "us"
    }
}
```

```
spark.read.json("record*.json").schema
```

```
root
 |-- event: struct (nullable = true)
 |    |-- message: string (nullable = true)
 |    |-- type: string (nullable = true)
 |    |-- ip: string (nullable = true)
 |    |-- browser: string (nullable = true)
 |    |-- os: string (nullable = true)
 |    |-- country: string (nullable = true)
 |-- host: string (nullable = true)
 |-- source: string (nullable = true)
 |-- time: long (nullable = true)
```

databricks

# JSON Generic Specified Schema

Print Schema

```
spark.read.json("record*.json").printSchema
```

StructType => MapType

```
customSchema = StructType([
    StructField("time", TimestampType(),True),
    StructField("host", StringType(),True),
    StructField("source", StringType(),True),
    StructField ("event",
             MapType(StringType(), StringType()))
    ])
```

Specify Schema

```
spark.read.schema(customSchema).json("record*.json")
```

## Specify Schemas!

### Faster

- No scan to infer the schema

### More Flexible

- Easily handle evolving or variable length fields

### More Robust

- Handle type errors on ETL vs query

databricks

# Agenda

1. **ETL:** Why ETL?
2. **Schemas**
3. **Metadata:** Best Practices FS Metadata
4. **Performance:** Tips & Tricks
5. **Error Handling**

databricks

## Filesystem Metadata Management

### Common Source FS Metadata

- Partitioned by arrival time (“path/yyyy/MM/dd/HH”)
- Many small files (kBs - 10s MB)
- Too little data per partition

```
...myRawData/2017/01/29/01
...myRawData/2017/01/29/02
```

databricks

# Backfill - Naive Approach

## Backfill

- Use only wildcards

```
df = spark.read.json("myRawData/2016/*/*/*/*")
.
.
.
df.write.partitionBy($“date”).parquet("myParquetData")
```

## Why this is a poor approach

- Spark (currently) is not aware of the existing partitions (yyyy/MM/dd/HH)
- Expensive full shuffle

# Backfill - Naive Approach

## Backfill

- Use only wildcards

```
df = spark.read.json("myRawData/2016/*/*/*/*")
.
.
.
df.write.partitionBy($“date”).parquet("myParquetData")
```

## Why this is a poor approach

- Spark (currently) is not aware of the existing partitions (yyyy/MM/dd/HH)
- Expensive full shuffle

databricks

## Backfill - Scaleable Approach

List of Paths

- Create a list of paths to backfill
- Use FS ls commands

Iterate over list to backfill

- Backfill each path

Operate in parallel

- Use Multiple Threads

  Scala .par

  Python multithreading

```
def convertToParquet (path:String) = {
  val df = spark.read.json(path)
          .
          .
  df.coalesce(20).write.mode("overwrite").save()
}
```

~~`dirList.foreach(x => convertToParquet(x))`~~

```
dirList.par.foreach(x => convertToParquet(x))
```

databricks

# Source Path Considerations

## Directory

- Specify Date instead of Year, Month, Day

## Block Sizes

- Read: Parquet Larger is Okay
- Write: 500MB-1GB

## Blocks / GBs per Partition

- Beware of Over Partitioning
- ~30GB per Partition

# Agenda

1. **ETL:** Why ETL?
2. **Schemas**
3. **Metadata:** Best Practices FS Metadata
4. **Performance:** Tips & Tricks
5. **Error Handling**

## Performance Optimizations

- Understand how Spark interprets Null Values
  - `nullValue`: specifies a string that indicates a null value, any fields matching this string will be set as nulls in the DataFrame

```
df = spark.read.options(header='true', inferschema='true', \
                        nullValue='\\N').csv('people.csv')
```

- Spark can understand it's own null data type
  - Users must translate their null types to Spark's native null type

databricks

# Test Data

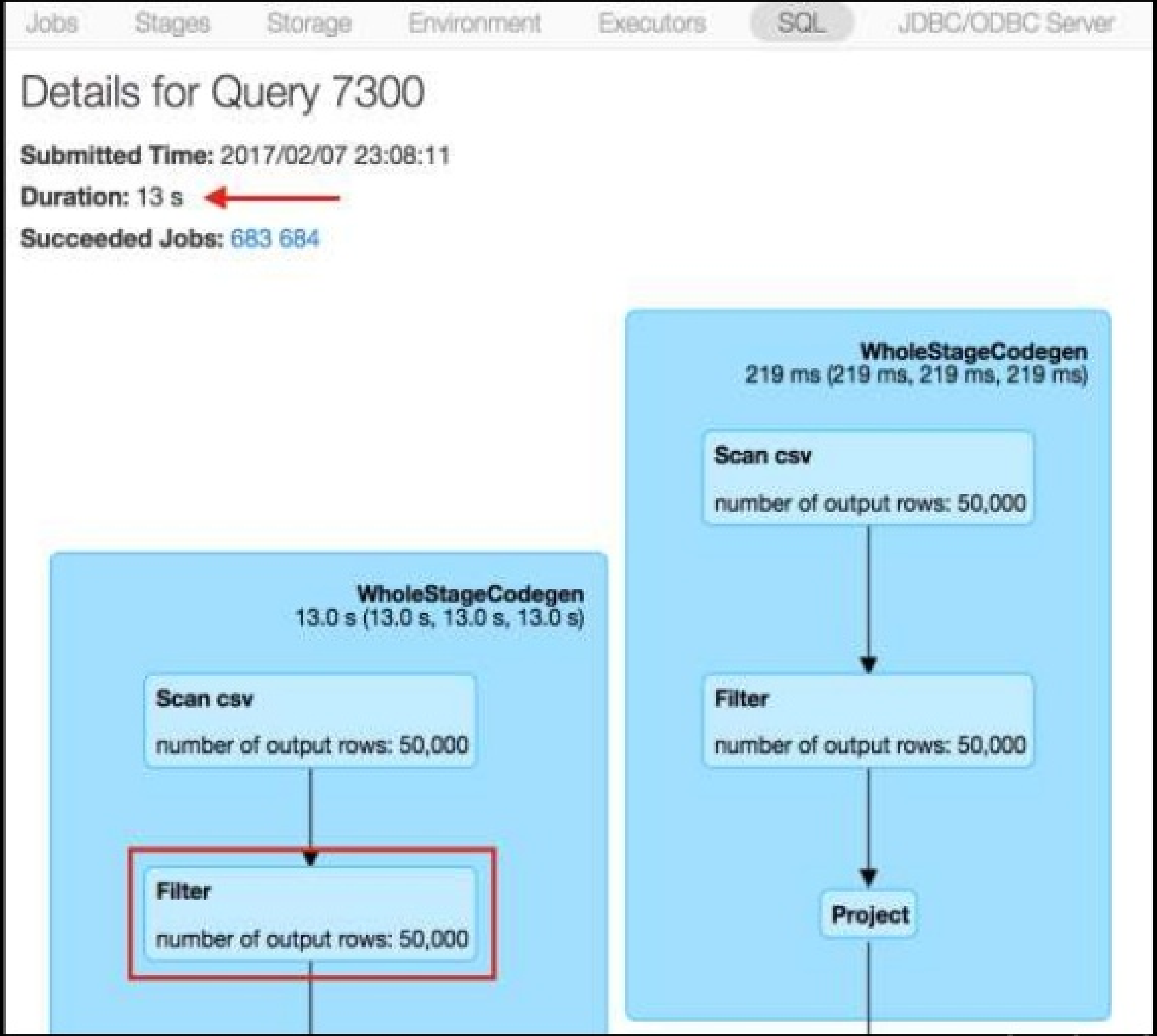
| *id* | *name* |
|---|---|
|  | foo |
| 2 | bar |
|  | bar |
| 11 | foo |
|  | bar |

| *id* | *name* |
|---|---|
| null | bar |
| null | bar |
| 3 | foo |
| 15 | foo |
| 2 | foo |

databricks

# Performance: Join Key

- Spark's WholeStageCodeGen will attempt to filter null values within the join key
- Ex: Empty String Test Data


Jobs
Stages
Storage
Environment
Executors
SQL
JDBC/ODBC Server

Details for Query 7300

**Submitted Time:** 2017/02/07 23:08:11

**Duration:** 13 s

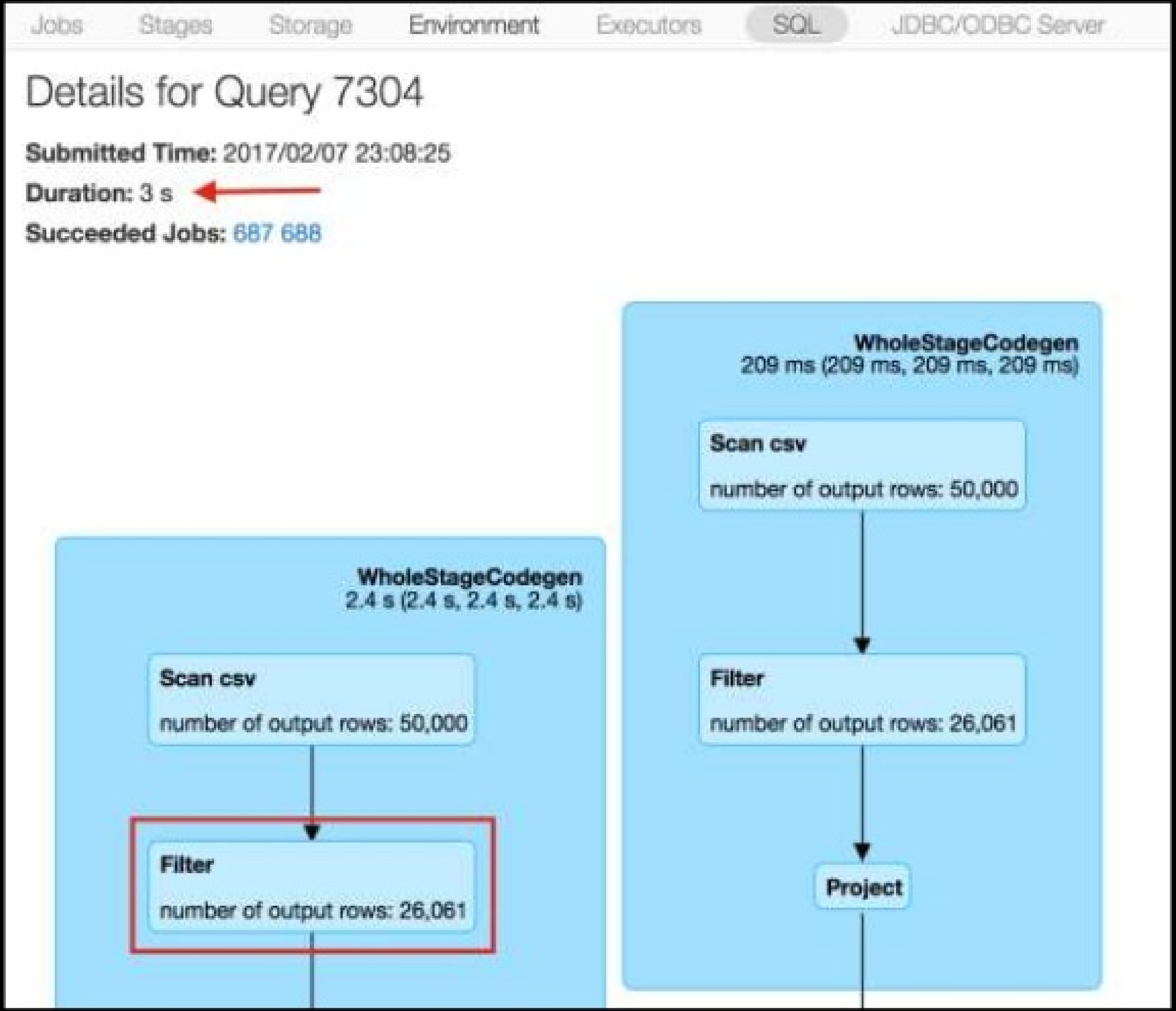
**Succeeded Jobs:** 683 684

WholeStageCodegen
219 ms (219 ms, 219 ms, 219 ms)
Scan csv
number of output rows: 50,000
Filter
number of output rows: 50,000
Project
WholeStageCodegen
13.0 s (13.0 s, 13.0 s, 13.0 s)
Scan csv
number of output rows: 50,000
Filter
number of output rows: 50,000

databricks

# Performance: Join Key

- The run time decreased from 13 seconds to 3 seconds
- Ex: Null Values for Empty Strings



Jobs
Stages
Storage
Environment
Executors
SQL
JDBC/ODBC Server

Details for Query 7304

**Submitted Time:** 2017/02/07 23:08:25

**Duration:** 3 s

**Succeeded Jobs:** 687 688

WholeStageCodegen
209 ms (209 ms, 209 ms, 209 ms)
Scan csv
number of output rows: 50,000
Filter
number of output rows: 26,061
Project
WholeStageCodegen
2.4 s (2.4 s, 2.4 s, 2.4 s)
Scan csv
number of output rows: 50,000
Filter
number of output rows: 26,061

databricks

# DataFrame Joins

```
df = spark.read.options(header='true').csv('/mnt/mwc/csv_join_empty')

df_duplicate = df.join(df, df['id'] == df['id'])

df_duplicate.printSchema()
```

(2) Spark Jobs

```
root
 |-- id: string (nullable = true)
 |-- name1: string (nullable = true)
 |-- id: string (nullable = true)
 |-- name2: string (nullable = true)
```

databricks

# DataFrame Joins

```
df = spark.read.options(header='true').csv('/mnt/mwc/csv_join_empty')

df_id = df.join(df, 'id')

df_id.printSchema()
```

(2) Spark Jobs

```
root
 |-- id: string (nullable = true)
 |-- name1: string (nullable = true)
 |-- name2: string (nullable = true)
```

Optimizing Apache Spark SQL Joins:
https://spark-summit.org/east-2017/events/optimizing-apache-spark-sql-joins/

databricks

# Agenda

1. **ETL:** Why ETL?
2. **Schemas**
3. **Metadata:** Best Practices FS Metadata
4. **Performance:** Tips & Tricks
5. **Error Handling**

databricks

## Error Handling: Corrupt Records

```
display(json_df)
```

| _corrupt_records | eventType | host | time |
|---|---|---|---|
| null | test1 | my.app.com | 1486166400 |
| null | test2 | my.app.com | 1486166402 |
| null | test3 | my.app2.com | 1486166404 |
| {"time": 1486166406, "host": "my2.app.com", "event_ } | null | null | null |
| null | test5 | my.app2.com | 1486166408 |

databricks

## Error Handling: UDFs

```
Py4JJavaError: An error occurred while calling o270.showString.

: org.apache.spark.SparkException: Job aborted due to stage failure: Task 0 in stage 59.0 failed 4 times, most
recent failure: Lost task 0.3 in stage 59.0 (TID 607, 172.128.245.47, executor 1):
org.apache.spark.api.python.PythonException: Traceback (most recent call last):

...

  File "/databricks/spark/python/pyspark/worker.py", line 92, in <lambda>

    mapper = lambda a: udf(*a)

...

  File "<ipython-input-10-b7bf56c9b155>", line 7, in add_minutes

AttributeError: type object 'datetime.datetime' has no attribute 'timedelta'

            at org.apache.spark.api.python.PythonRunner$$anon$1.read(PythonRDD.scala:193)
```

## Error Handling: UDFs

- Spark UDF Example

```
from datetime import date

from datetime import datetime, timedelta

def add_minutes(start_date, minutes_to_add):

    return datetime.combine(b, datetime.min.time()) + timedelta(minutes=long(minutes_to_add))
```

- Best Practices
  - Verify the input data types from the DataFrame
  - Sample data to verify the UDF
  - Add test cases

# Error Handling: InputFiles

- Identify the records input source file

```
from pyspark.sql.functions import *

df = df.withColumn("fname", input_file_name())

display(spark.sql("select *, input_file_name() from companies"))
```

| city | zip | fname |
|---|---|---|
| LITTLETON | 80127 | dbfs:/user/hive/warehouse/companies/part-r-00000-8cf2a23f-f3b5-4378-b7b0-72913fbb7414.gz.parquet |
| BOSTON | 02110 | dbfs:/user/hive/warehouse/companies/part-r-00000-8cf2a23f-f3b5-4378-b7b0-72913fbb7414.gz.parquet |
| NEW YORK | 10019 | dbfs:/user/hive/warehouse/companies/part-r-00000-8cf2a23f-f3b5-4378-b7b0-72913fbb7414.gz.parquet |
| SANTA CLARA | 95051 | dbfs:/user/hive/warehouse/companies/part-r-00000-8cf2a23f-f3b5-4378-b7b0-72913fbb7414.gz.parquet |

databricks

# Thanks

## Enjoy The Snow

https://databricks.com/try-databricks

databricks