# ETL Part 1: Data Extraction

In this course data engineers access data where it lives and then apply data extraction best practices, including schemas, corrupt record handling, and parallelized code. By the end of this course, you will extract data from multiple sources, use schema inference and apply user-defined schemas, and navigate Databricks and Spark documents to source solutions.

## Lessons

1. Course Overview and Setup
2. ETL Process Overview
3. Connecting to Cloud Data Storage
4. Connecting to JDBC
5. Applying Schemas to JSON Data
6. Corrupt Record Handling
7. Loading Data and Productionalizing

## ETL with Databricks and Spark

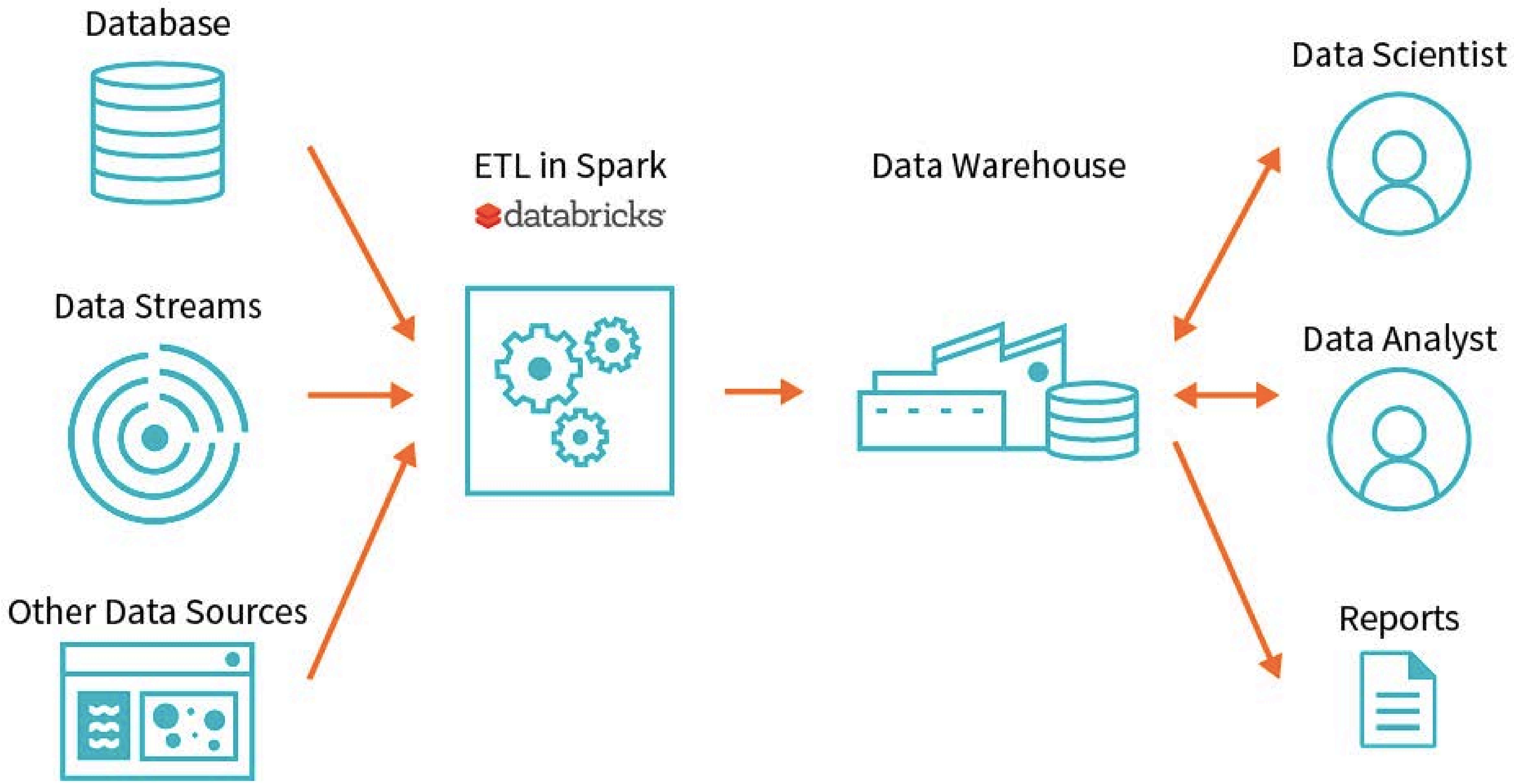
The **extract, transform, load (ETL)** process takes data from one or more sources, transforms it, normally by adding structure, and then loads it into a target database.

A common ETL job takes log files from a web server, parses out pertinent fields so it can be readily queried, and then loads it into a database.

ETL may seem simple: applying structure to data so it's in a desired form. However, the complexity of ETL is in the details. Data Engineers building ETL pipelines must understand and apply the following concepts:

- Optimizing data formats and connections
- Determining the ideal schema
- Handling corrupt records
- Automating workloads

This course addresses these concepts.


Database
Data Streams
Other Data Sources
ETL in Spark
databricks
Data Warehouse
Data Scientist
Data Analyst
Reports

# ETL Process Overview

Apache Spark™ and Databricks® allow you to create an end-to-end *extract, transform, load (ETL)* pipeline.

## In this lesson you:

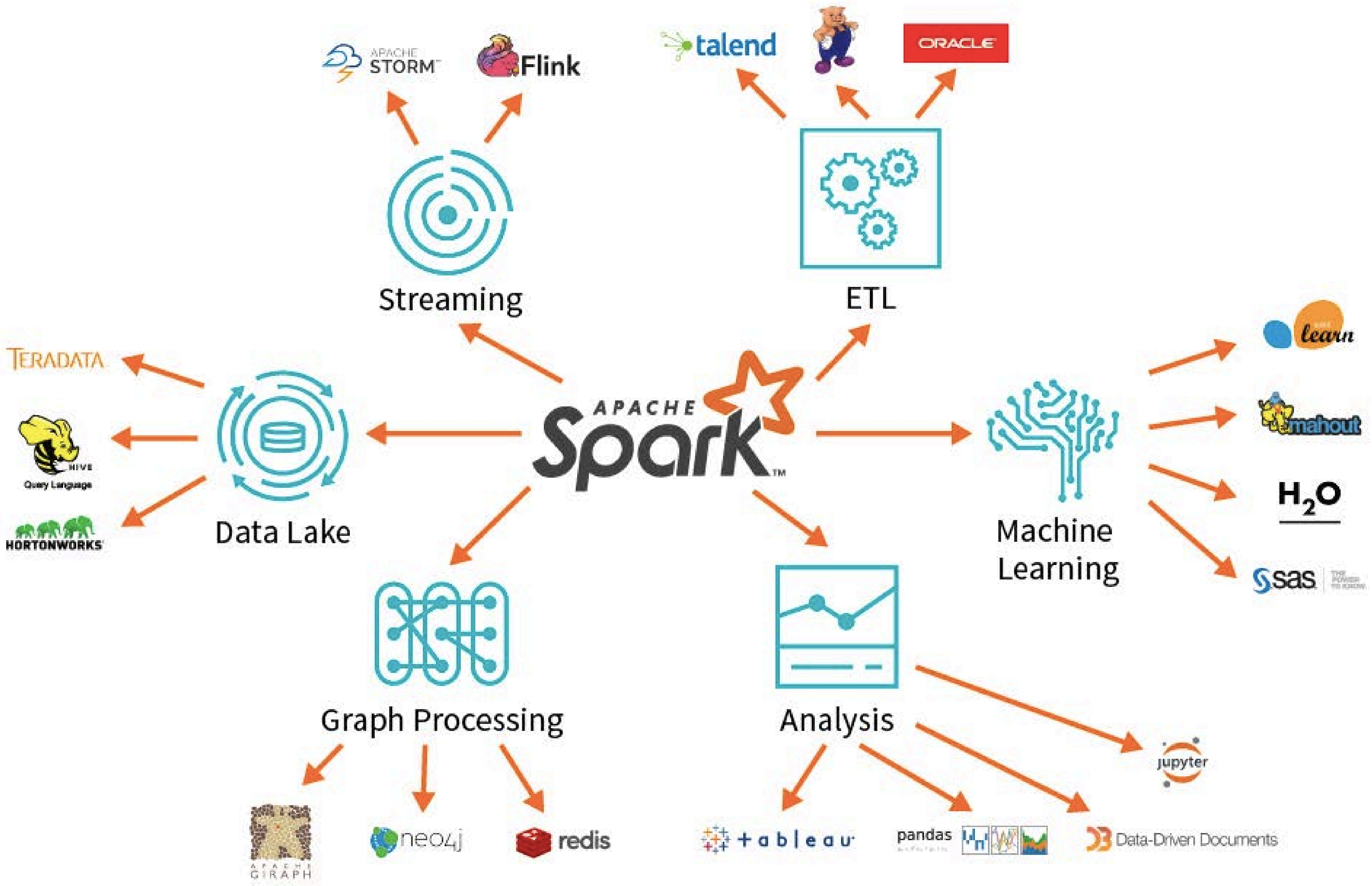
- Create a basic end-to-end ETL pipeline
- Demonstrate the Spark approach to ETL pipelines

## The Spark Approach

Spark offers a compute engine and connectors to virtually any data source. By leveraging easily scaled infrastructure and accessing data where it lives, Spark addresses the core needs of a big data application.

These principles comprise the Spark approach to ETL, providing a unified and scalable approach to big data pipelines:

1. Databricks and Spark offer a **unified platform**
   - Spark on Databricks combines ETL, stream processing, machine learning, and collaborative notebooks.
   - Data scientists, analysts, and engineers can write Spark code in Python, Scala, SQL, and R.
2. Spark's unified platform is **scalable to petabytes of data and clusters of thousands of nodes**.
   - The same code written on smaller data sets scales to large workloads, often with only small changes.
3. Spark on Databricks decouples data storage from the compute and query engine.
   - Spark's query engine **connects to any number of data sources** such as S3, Azure Blob Storage, Redshift, and Kafka.
   - This **minimizes costs**; a dedicated cluster does not need to be maintained and the compute cluster is **easily updated to the latest version** of Spark.


APACHE STORM
Flink
talend
ORACLE
Streaming
ETL
TERADATA
HIVE Query Language
HORTONWORKS
Data Lake
APACHE Spark
learn
mahout
$H_2O$
sas
Machine Learning
Graph Processing
Analysis
jupyter
APACHE GIRAPH
neo4j
redis
tableau
pandas
Data-Driven Documents

## A Basic ETL Job

In this lesson you use web log files from the US Securities and Exchange Commission website to do a basic ETL for a day of server activity. You will extract the fields of interest and load them into persistent storage.

The Databricks File System (DBFS) is an HDFS-like interface to bulk data stores like Amazon's S3 and Azure's Blob storage service.

Pass the path `/mnt/training/EDGAR-Log-20170329/EDGAR-Log-20170329.csv` into `spark.read.csv` to access data stored in DBFS. Use the header option to specify that the first line of the file is the header.

Cmd 10

```
path = "/mnt/training/EDGAR-Log-20170329/EDGAR-Log-20170329.csv"

logDF = (spark
  .read
  .option("header", True)
  .csv(path)
  .sample(withReplacement=False, fraction=0.3, seed=3) # using a sample to reduce data size
)

display(logDF)
```

The Databricks File System (DBFS) is an HDFS-like interface to bulk data stores like Amazon's S3 and Azure's Blob storage service.

Pass the path `/mnt/training/EDGAR-Log-20170329/EDGAR-Log-20170329.csv` into `spark.read.csv` to access data stored in DBFS. Use the header option to specify that the first line of the file is the header.

Cmd 10

```python
path = "/mnt/training/EDGAR-Log-20170329/EDGAR-Log-20170329.csv"

logDF = (spark
  .read
  .option("header", True)
  .csv(path)
  .sample(withReplacement=False, fraction=0.3, seed=3) # using a sample to reduce data size
)

display(logDF)
```

▸ (2) Spark Jobs

▸ logDF: pyspark.sql.dataframe.DataFrame = [ip: string, date: string ... 13 more fields]

| ip | date | time | zone | cik | accession | extention | code | size | idx | norefer | noagent | find | crawler | browser |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 101.71.41.ihh | 2017-03-29 | 00:00:00 | 0.0 | 1437491.0 | 0001245105-17-000052 | xslF345X03/primary_doc.xml | 301.0 | 687.0 | 0.0 | 0.0 | 0.0 | 10.0 | 0.0 | null |
| 104.196.240.dda | 2017-03-29 | 00:00:00 | 0.0 | 1270985.0 | 0001188112-04-001037 | .txt | 200.0 | 7619.0 | 0.0 | 0.0 | 0.0 | 10.0 | 0.0 | null |
| 107.23.85.jfd | 2017-03-29 | 00:00:00 | 0.0 | 1059376.0 | 0000905148-07-006108 | -index.htm | 200.0 | 2727.0 | 1.0 | 0.0 | 0.0 | 10.0 | 0.0 | null |
| 107.23.85.jfd | 2017-03-29 | 00:00:00 | 0.0 | 1059376.0 | 0000905148-08-001993 | -index.htm | 200.0 | 2710.0 | 1.0 | 0.0 | 0.0 | 10.0 | 0.0 | null |
| 107.23.85.jfd | 2017-03-29 | 00:00:00 | 0.0 | 1059376.0 | 0001104659-09-046963 | -index.htm | 200.0 | 2715.0 | 1.0 | 0.0 | 0.0 | 10.0 | 0.0 | null |
| 107.23.85.jfd | 2017-03-29 | 00:00:00 | 0.0 | 1364986.0 | 0000914121-06-002243 | -index.htm | 200.0 | 2786.0 | 1.0 | 0.0 | 0.0 | 10.0 | 0.0 | null |
| 107.23.85.jfd | 2017-03-29 | 00:00:00 | 0.0 | 1364986.0 | 0000914121-06-002251 | -index.htm | 200.0 | 2784.0 | 1.0 | 0.0 | 0.0 | 10.0 | 0.0 | null |

Next, review the server-side errors, which have error codes in the 500s.

Cmd 12

```python
from pyspark.sql.functions import col

serverErrorDF = (logDF
  .filter((col("code") >= 500) & (col("code") < 600))
  .select("date", "time", "extention", "code")
)

display(serverErrorDF)
```

▸ (2) Spark Jobs

▸ serverErrorDF: pyspark.sql.dataframe.DataFrame = [date: string, time: string ... 2 more fields]

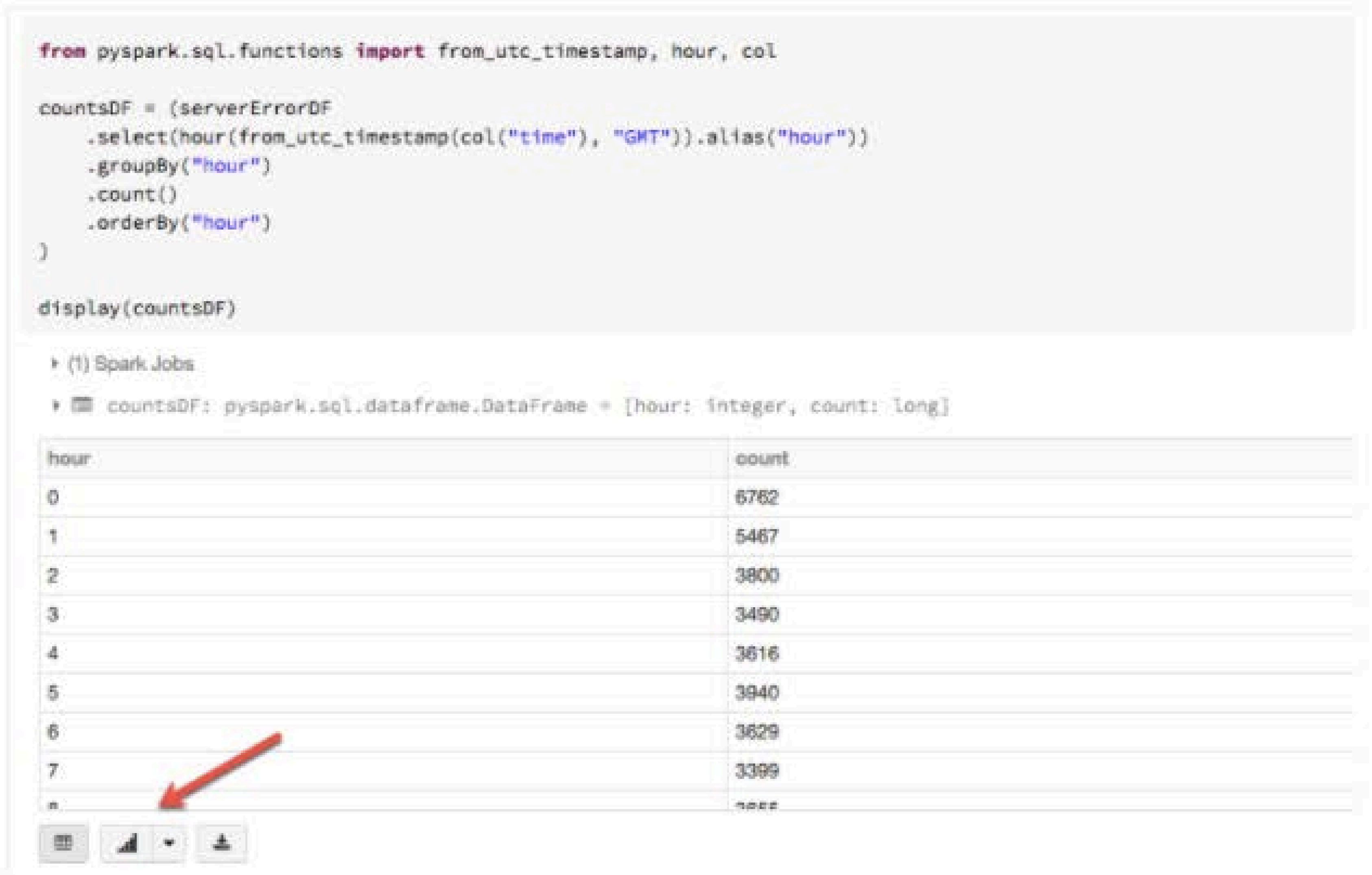
| date | time | extention | code |
|---|---|---|---|
| 2017-03-29 | 00:00:12 | .txt | 503.0 |
| 2017-03-29 | 00:00:16 | -index.htm | 503.0 |
| 2017-03-29 | 00:00:24 | -index.htm | 503.0 |
| 2017-03-29 | 00:00:44 | -index.htm | 503.0 |
| 2017-03-29 | 00:01:01 | -index.htm | 503.0 |
| 2017-03-29 | 00:01:01 | -index.htm | 503.0 |

## Data Validation

One aspect of ETL jobs is to validate that the data is what you expect. This includes:

- Approximately the expected number of records
- The expected fields are present
- No unexpected missing values

Take a look at the server-side errors by hour to confirm the data meets your expectations. Visualize it by selecting the bar graph icon once the table is displayed.

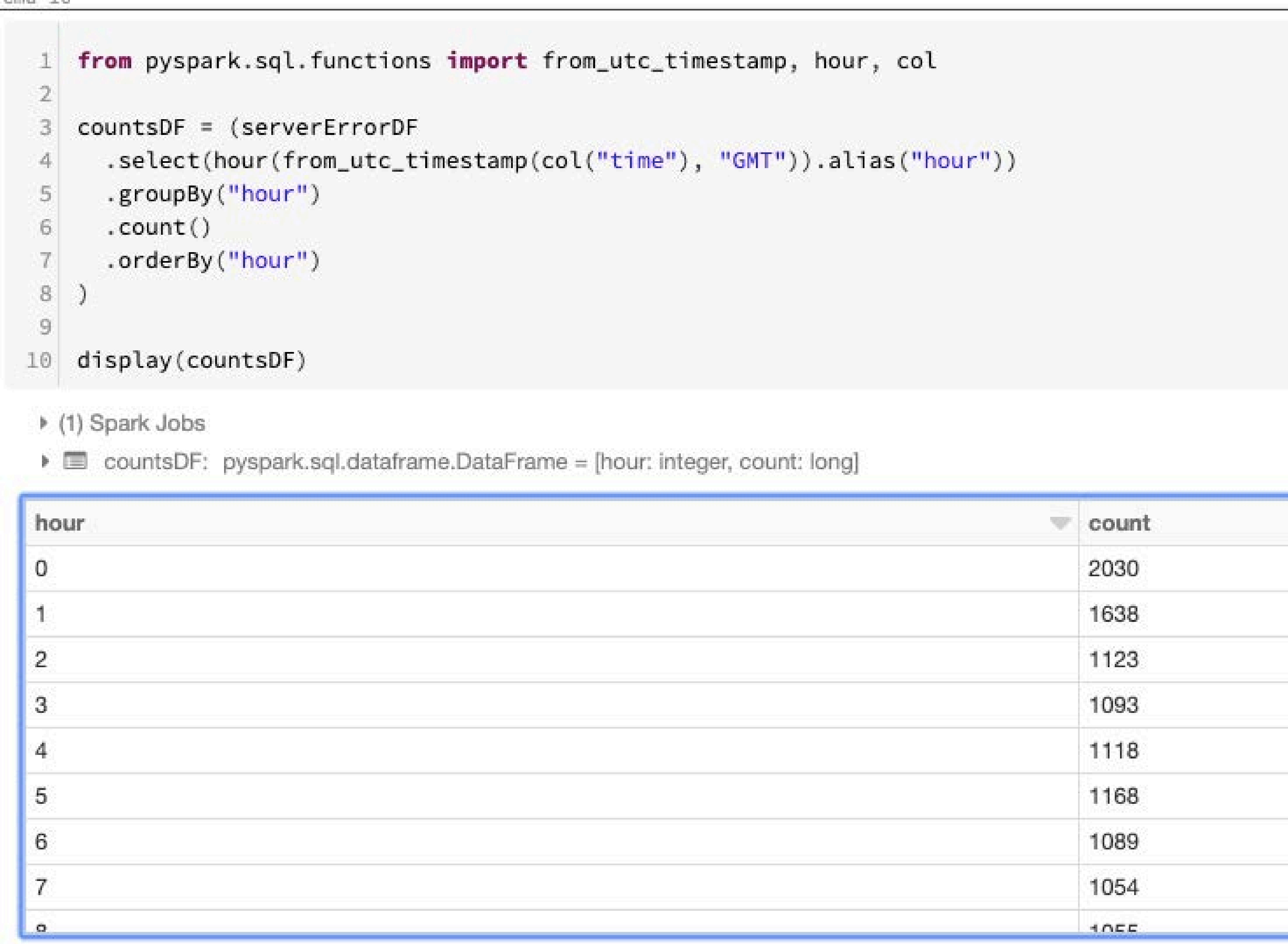


```
from pyspark.sql.functions import from_utc_timestamp, hour, col

countsDF = (serverErrorDF
    .select(hour(from_utc_timestamp(col("time"), "GMT")).alias("hour"))
    .groupBy("hour")
    .count()
    .orderBy("hour")
)

display(countsDF)
```

▸ (1) Spark Jobs

▸ countsDF: pyspark.sql.dataframe.DataFrame = [hour: integer, count: long]

| hour | count |
|---|---|
| 0 | 6762 |
| 1 | 5467 |
| 2 | 3800 |
| 3 | 3490 |
| 4 | 3616 |
| 5 | 3940 |
| 6 | 3629 |
| 7 | 3399 |
| 8 | [illegible] |



```python
from pyspark.sql.functions import from_utc_timestamp, hour, col

countsDF = (serverErrorDF
  .select(hour(from_utc_timestamp(col("time"), "GMT")).alias("hour"))
  .groupBy("hour")
  .count()
  .orderBy("hour")
)

display(countsDF)
```

▸ (1) Spark Jobs

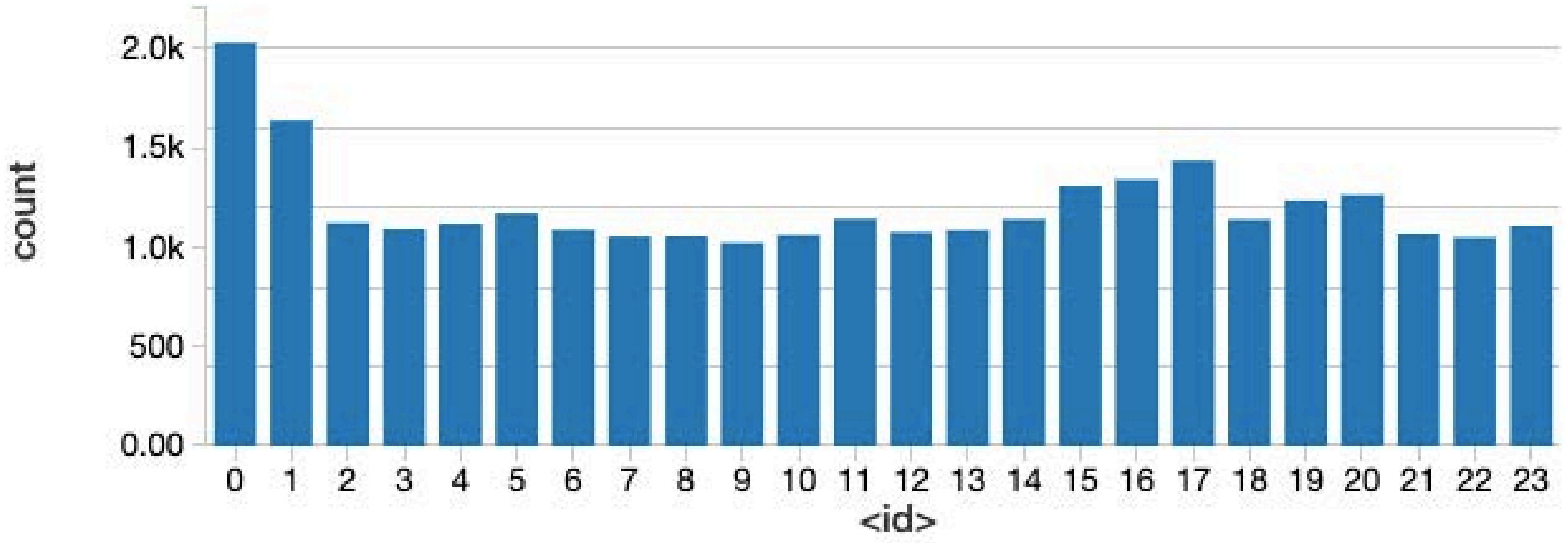
▸ countsDF: pyspark.sql.dataframe.DataFrame = [hour: integer, count: long]

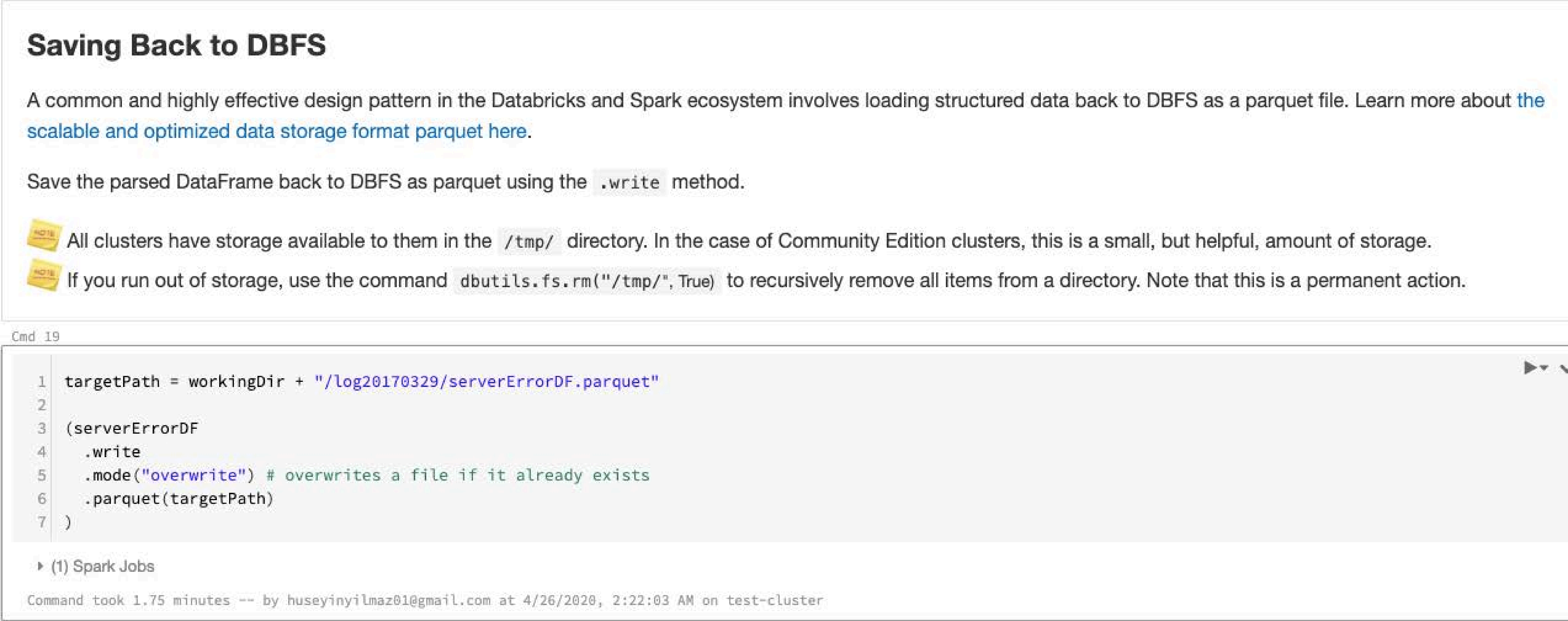
| hour | count |
|---|---|
| 0 | 2030 |
| 1 | 1638 |
| 2 | 1123 |
| 3 | 1093 |
| 4 | 1118 |
| 5 | 1168 |
| 6 | 1089 |
| 7 | 1054 |

```python
from pyspark.sql.functions import from_utc_timestamp, hour, col

countsDF = (serverErrorDF
  .select(hour(from_utc_timestamp(col("time"), "GMT")).alias("hour"))
  .groupBy("hour")
  .count()
  .orderBy("hour")
)

display(countsDF)
```

▸ (1) Spark Jobs

▸ countsDF: pyspark.sql.dataframe.DataFrame = [hour: integer, count: long]





## Saving Back to DBFS

A common and highly effective design pattern in the Databricks and Spark ecosystem involves loading structured data back to DBFS as a parquet file. Learn more about the scalable and optimized data storage format parquet here.

Save the parsed DataFrame back to DBFS as parquet using the `.write` method.

All clusters have storage available to them in the `/tmp/` directory. In the case of Community Edition clusters, this is a small, but helpful, amount of storage.

If you run out of storage, use the command `dbutils.fs.rm("/tmp/", True)` to recursively remove all items from a directory. Note that this is a permanent action.

Cmd 19

```
targetPath = workingDir + "/log20170329/serverErrorDF.parquet"

(serverErrorDF
  .write
  .mode("overwrite") # overwrites a file if it already exists
  .parquet(targetPath)
)
```

▸ (1) Spark Jobs

```
Command took 1.75 minutes -- by huseyinyilmaz01@gmail.com at 4/26/2020, 2:22:03 AM on test-cluster
```

## Our ETL Pipeline

Here's what the ETL pipeline you just built looks like. In the rest of this course you will work with more complex versions of this general pattern.

| Code | Stage |
|---|---|
| `logDF = (spark` | Extract |
| `  .read` | Extract |
| `  .option("header", True)` | Extract |
| `  .csv()` | Extract |
| `)` | Extract |
| `serverErrorDF = (logDF` | Transform |
| `  .filter((col("code") >= 500) & (col("code") < 600))` | Transform |
| `  .select("date", "time", "extention", "code")` | Transform |
| `)` | Transform |
| `(serverErrorDF.write` | Load |
| `  .parquet())` | Load |

This is a distributed job, so it can easily scale to fit the demands of your data set.

# Exercise 1: Perform an ETL Job

Write a basic ETL script that captures the 20 most active website users and load the results to DBFS.

Cmd 23

## Step 1: Create a DataFrame of Aggregate Statistics

Create a DataFrame `ipCountDF` that uses `logDF` to create a count of each time a given IP address appears in the logs, with the counts sorted in descending order. The result should have two columns: `ip` and `count`.

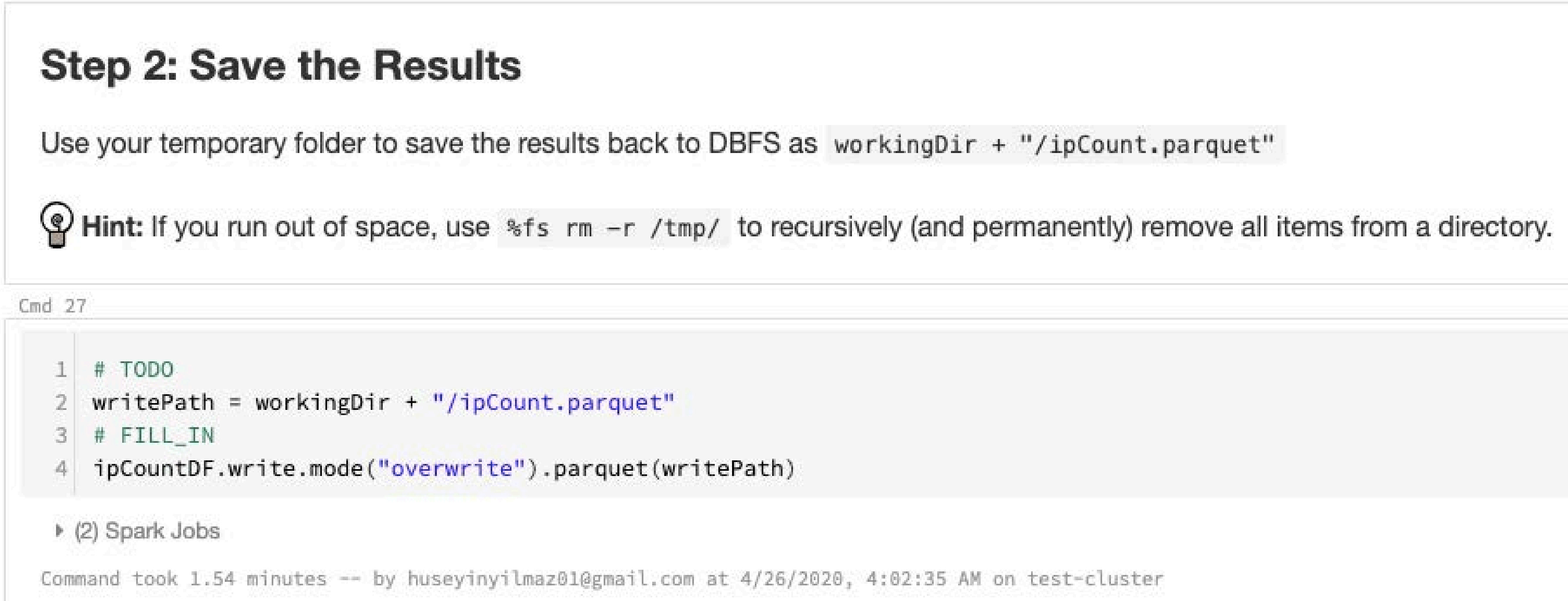
Cmd 24

```python
# TODO
from pyspark.sql.functions import desc
ipCountDF = (logDF
  .select(col("ip"))
  .groupBy("ip")
  .count()
  .orderBy(desc("count"))
)

display(ipCountDF)
```

(1) Spark Jobs

ipCountDF: pyspark.sql.dataframe.DataFrame = [ip: string, count: long]

| ip | count |
|---|---|
| 213.152.28.bhe | 518548 |
| 158.132.91.haf | 497361 |
| 117.91.6.caf | 239912 |
| 132.195.122.djf | 197267 |



## Step 2: Save the Results

Use your temporary folder to save the results back to DBFS as `workingDir + "/ipCount.parquet"`

**Hint:** If you run out of space, use `%fs rm -r /tmp/` to recursively (and permanently) remove all items from a directory.

Cmd 27

```
# TODO
writePath = workingDir + "/ipCount.parquet"
# FILL_IN
ipCountDF.write.mode("overwrite").parquet(writePath)
```

▸ (2) Spark Jobs

```
Command took 1.54 minutes -- by huseyinyilmaz01@gmail.com at 4/26/2020, 4:02:35 AM on test-cluster
```

## Review

**Question:** What does ETL stand for and what are the stages of the process?
**Answer:** ETL stands for `extract-transform-load`
1. *Extract* refers to ingesting data. Spark easily connects to data in a number of different sources.
2. *Transform* refers to applying structure, parsing fields, cleaning data, and/or computing statistics.
3. *Load* refers to loading data to its final destination, usually a database or data warehouse.

**Question:** How does the Spark approach to ETL deal with devops issues such as updating a software version?
**Answer:** By decoupling storage and compute, updating your Spark version is as easy as spinning up a new cluster. Your old code will easily connect to S3, the Azure Blob, or other storage. This also avoids the challenge of keeping a cluster always running, such as with Hadoop clusters.

**Question:** How does the Spark approach to data applications differ from other solutions?
**Answer:** Spark offers a unified solution to use cases that would otherwise need individual tools. For instance, Spark combines machine learning, ETL, stream processing, and a number of other solutions all with one technology.