Spark Developer Certification - Comprehensive Study Guide (python)

What is this?

While studying for the Spark certification exam (https://academy.databricks.com/exam/crt020-python) and going through various resources available online, I (https://www.linkedin.com/in/mdrakiburrahman/) thought it'd be worthwhile to put together a comprehensive knowledge dump that covers the entire syllabus end-to-end, serving as a Study Guide for myself and hopefully others.

Note that I used inspiration from the Spark Code Review guide (https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/3249 - but whereas that covers a subset of the coding aspects only, I aimed for this to be more of a *comprehensive, one stop resource geared towards passing the exam*.

Awesome Resources/References used throughout this guide

References

- **Spark Code Review used for inspiration**: https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/32 (https://databricks-prod-cloudfront.cloud.databricks.com/public/4027ec902e239c93eaaa8714f173bcfc/32
- **Syllabus**: https://academy.databricks.com/exam/crt020-python (https://academy.databricks.com/exam/crt020-python)
- **Data Scientists Guide to Apache Spark**: https://drive.google.com/open?id=17KMSIlwgMvQ8cuvTbcwznnLB6-4Oen9T (https://drive.google.com/open?

id=17KMSllwgMvQ8cuvTbcwznnLB6-4Oen9T)

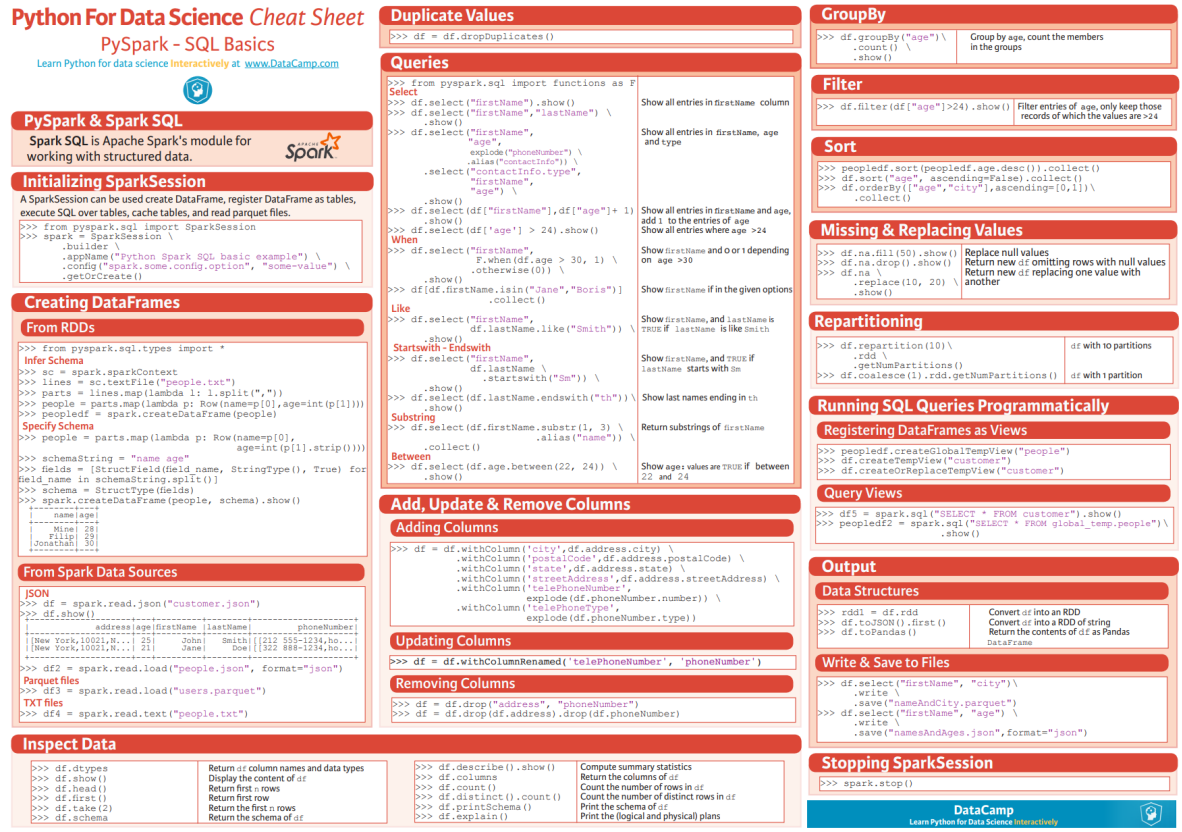
- **JVM Overview**: https://www.javaworld.com/article/3272244/what-is-the-jvm-introducing-the-java-virtual-machine.html (https://www.javaworld.com/article/3272244/what-is-the-jvm-introducing-the-java-virtual-machine.html)
- **Spark Runtime Architecture Overview:** https://freecontent.manning.com/running-spark-an-overview-of-sparks-runtime-architecture (https://freecontent.manning.com/running-spark-an-overview-of-sparks-runtime-architecture)
- **Spark Application Overview:** https://docs.cloudera.com/documentation/enterprise/5-6-x/topics/cdh_ig_spark_apps.html (https://docs.cloudera.com/documentation/enterprise/5-6-x/topics/cdh_ig_spark_apps.html)
- **Spark Architecture Overview:** http://queirozf.com/entries/spark-architecture-overview-clusters-jobs-stages-tasks-etc (http://queirozf.com/entries/spark-architecture-overview-clusters-jobs-stages-tasks-etc)
- **Mastering Apache Spark:** https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-scheduler-Stage.html (https://jaceklaskowski.gitbooks.io/mastering-apache-spark/spark-scheduler-Stage.html)
- **Manually create DFs:** https://medium.com/@mrpowers/manually-creating-spark-dataframes-b14dae906393 (https://medium.com/@mrpowers/manually-creating-spark-dataframes-b14dae906393)
- **PySpark SQL docs:** https://spark.apache.org/docs/latest/api/python/pyspark.sql.html (https://spark.apache.org/docs/latest/api/python/pyspark.sql.html)
- **Introduction to DataFrames:** https://docs.databricks.com/spark/latest/dataframes-datasets/introduction-to-dataframes-python.html (https://docs.databricks.com/spark/latest/dataframes-datasets/introduction-to-dataframes-python.html)
- **PySpark UDFs:** https://changhsinlee.com/pyspark-udf/ (https://changhsinlee.com/pyspark-udf/)
- **ORC File:** https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC (https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC)

- **SQL Server Stored Procedures from Databricks:** https://datathirst.net/blog/2018/10/12/executing-sql-server-stored-procedures-on-databricks-pyspark (https://datathirst.net/blog/2018/10/12/executing-sql-server-stored-procedures-on-databricks-pyspark)
- **Repartition vs Coalesce:** https://stackoverflow.com/questions/31610971/spark-repartition-vs-coalesce (https://stackoverflow.com/questions/31610971/spark-repartition-vs-coalesce)
- **Partitioning by Columns:** https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-dynamic-partition-inserts.html (https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-dynamic-partition-inserts.html)
- **Bucketing:** https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-bucketing.html (https://jaceklaskowski.gitbooks.io/mastering-spark-sql/spark-sql-bucketing.html)
- **PySpark GroupBy and Aggregate Functions:** https://hendra-herviawan.github.io/pyspark-groupby-and-aggregate-functions.html (https://hendra-herviawan.github.io/pyspark-groupby-and-aggregate-functions.html)
- **Spark Quickstart:** https://spark.apache.org/docs/latest/quick-start.html (https://spark.apache.org/docs/latest/quick-start.html)
- **Spark Caching - 1:** https://unraveldata.com/to-cache-or-not-to-cache/ (https://unraveldata.com/to-cache-or-not-to-cache/)
- **Spark Caching - 2:** https://stackoverflow.com/questions/45558868/where-does-df-cache-is-stored (https://stackoverflow.com/questions/45558868/where-does-df-cache-is-stored)
- **Spark Caching - 3:** https://changhsinlee.com/pyspark-dataframe-basics/ (https://changhsinlee.com/pyspark-dataframe-basics/)
- **Spark Caching - 4:** http://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence (http://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence)
- **Spark Caching - 5:** https://www.tutorialspoint.com/pyspark/pyspark_storagelevel.htm (https://www.tutorialspoint.com/pyspark/pyspark_storagelevel.htm)
- **Spark Caching - 6:** https://spark.apache.org/docs/2.1.2/api/python/_modules/pyspark/storagelevel.h (https://spark.apache.org/docs/2.1.2/api/python/_modules/pyspark/storagelevel.h

- **Spark SQL functions examples:** https://spark.apache.org/docs/2.3.0/api/sql/index.html (https://spark.apache.org/docs/2.3.0/api/sql/index.html)
- **Spark Built-in Higher Order Functions Examples:** https://docs.databricks.com/_static/notebooks/apache-spark-2.4-functions.html (https://docs.databricks.com/_static/notebooks/apache-spark-2.4-functions.html)
- **Spark SQL Timestamp conversion:** https://docs.databricks.com/_static/notebooks/timestamp-conversion.html (https://docs.databricks.com/_static/notebooks/timestamp-conversion.html)
- **RegEx Tutorial:** https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285 (https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285)
- **Rank VS Dense Rank:** https://stackoverflow.com/questions/44968912/difference-in-dense-rank-and-row-number-in-spark (https://stackoverflow.com/questions/44968912/difference-in-dense-rank-and-row-number-in-spark)
- **SparkSQL Windows:** https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html (https://databricks.com/blog/2015/07/15/introducing-window-functions-in-spark-sql.html)
- **Spark Certification Study Guide:** https://github.com/vivek-bombatkar/Databricks-Apache-Spark-2X-Certified-Developer (https://github.com/vivek-bombatkar/Databricks-Apache-Spark-2X-Certified-Developer)

Resources

PySpark Cheatsheet



1. Spark Architecture Components

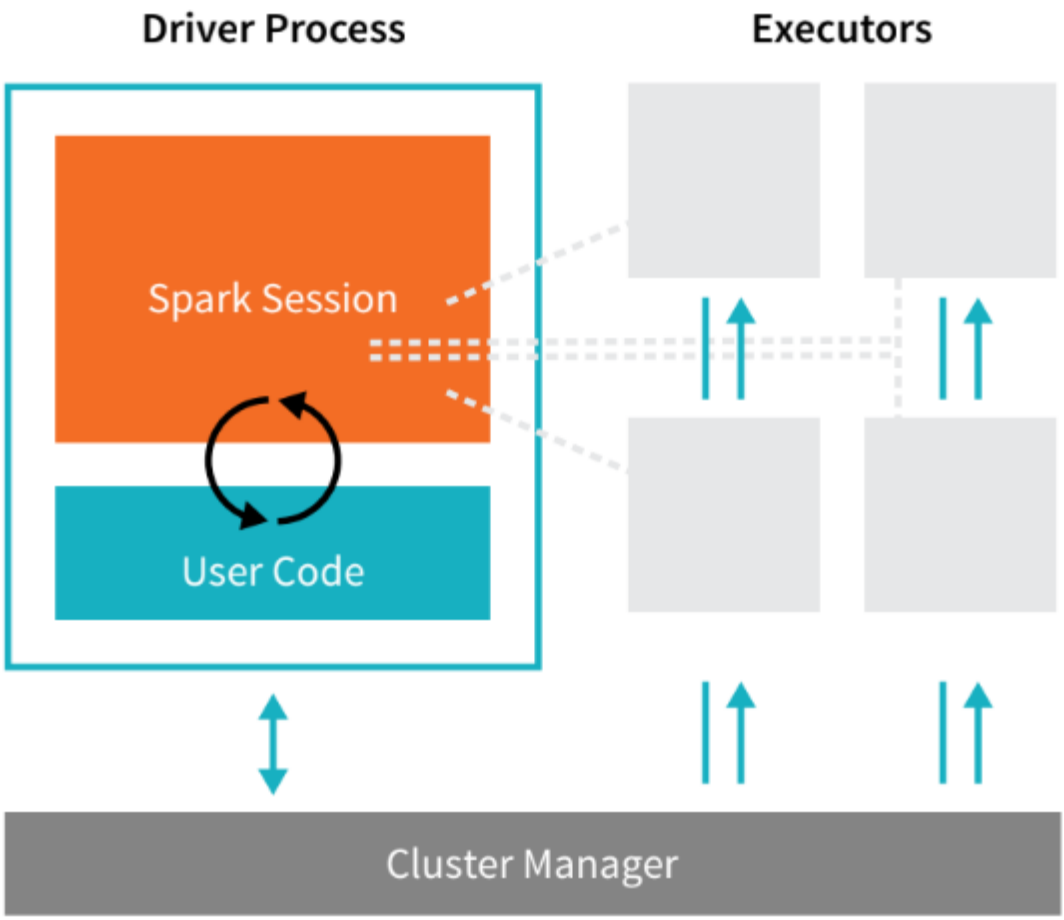
Candidates are expected to be familiar with the following architectural components and their relationship to each other:

Spark Basic Architecture

A *cluster*, or group of machines, pools the resources of many machines together allowing us to use all the cumulative resources as if they were one. Now a group of machines sitting somewhere alone is not powerful, you need a framework to coordinate work across them. **Spark** is a tailor-made engine exactly for this, managing and coordinating the execution of tasks on data across a cluster of computers.

The cluster of machines that Spark will leverage to execute tasks will be managed by a cluster manager like Spark's Standalone cluster manager, ***YARN*** - **Y**et **A**nother **R**esource **N**egotiator, or ***Mesos*** (http://mesos.apache.org/). We then submit Spark Applications to these cluster managers which will grant resources to our application so that we can complete our work.

Spark Applications



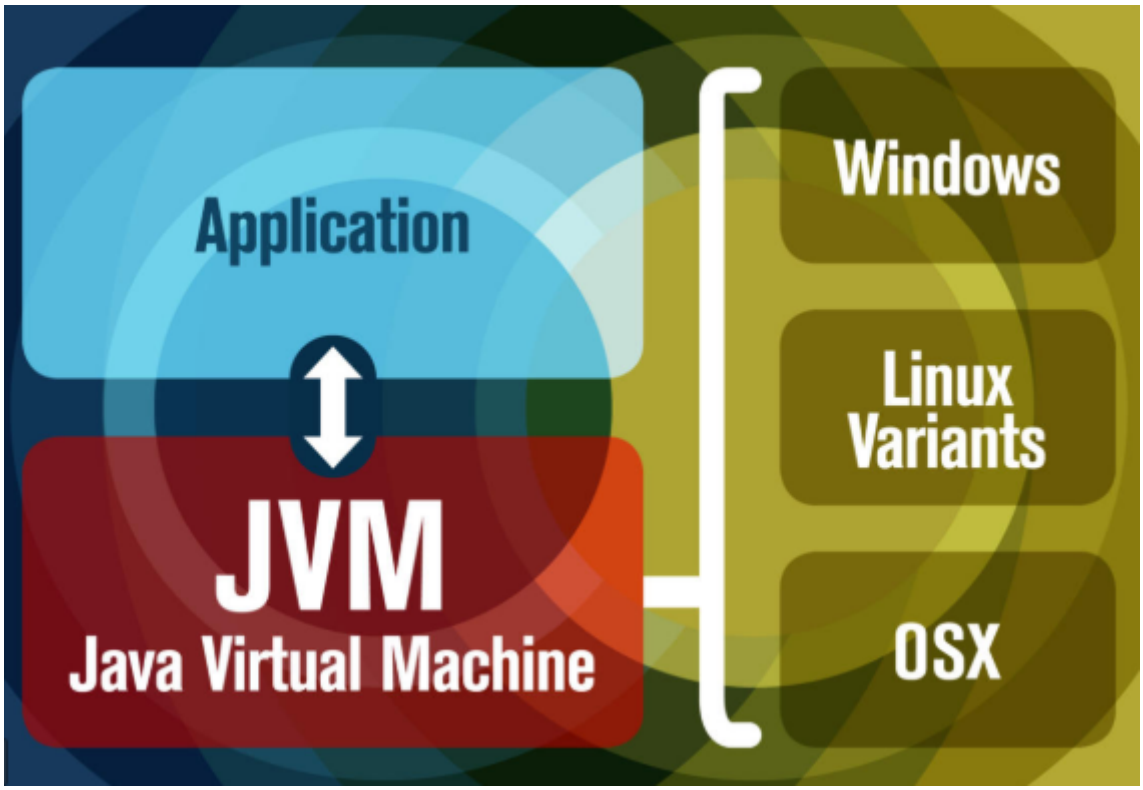
Spark Applications consist of a **driver** process and a set of **executor** processes. In the illustration we see above, our driver is on the left and four executors on the right.

What is a JVM?

The JVM manages system memory and provides a portable execution environment for Java-based applications

Technical definition: The JVM is the specification for a software program that executes code and provides the runtime environment for that code.
Everyday definition: The JVM is how we run our Java programs. We configure the JVM's settings and then rely on it to manage program resources during execution.

The **Java Virtual Machine (JVM)** is a program whose purpose is to execute other programs.

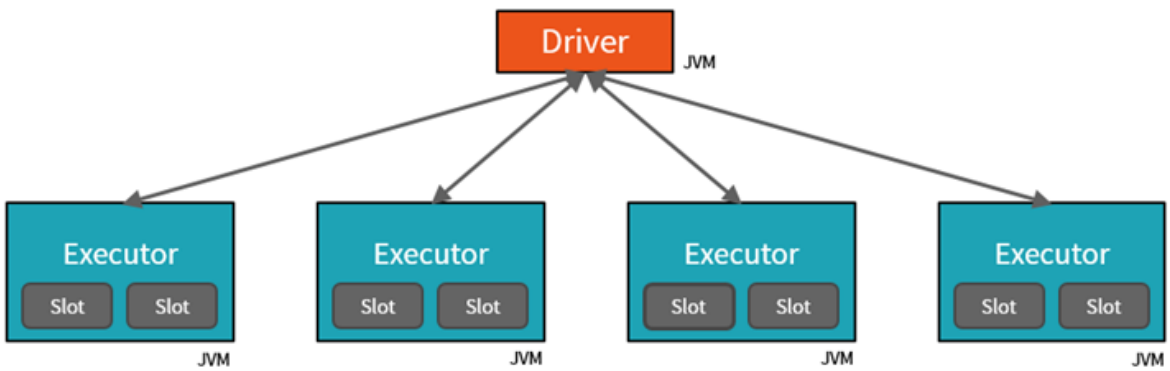


The JVM has **two primary functions**:

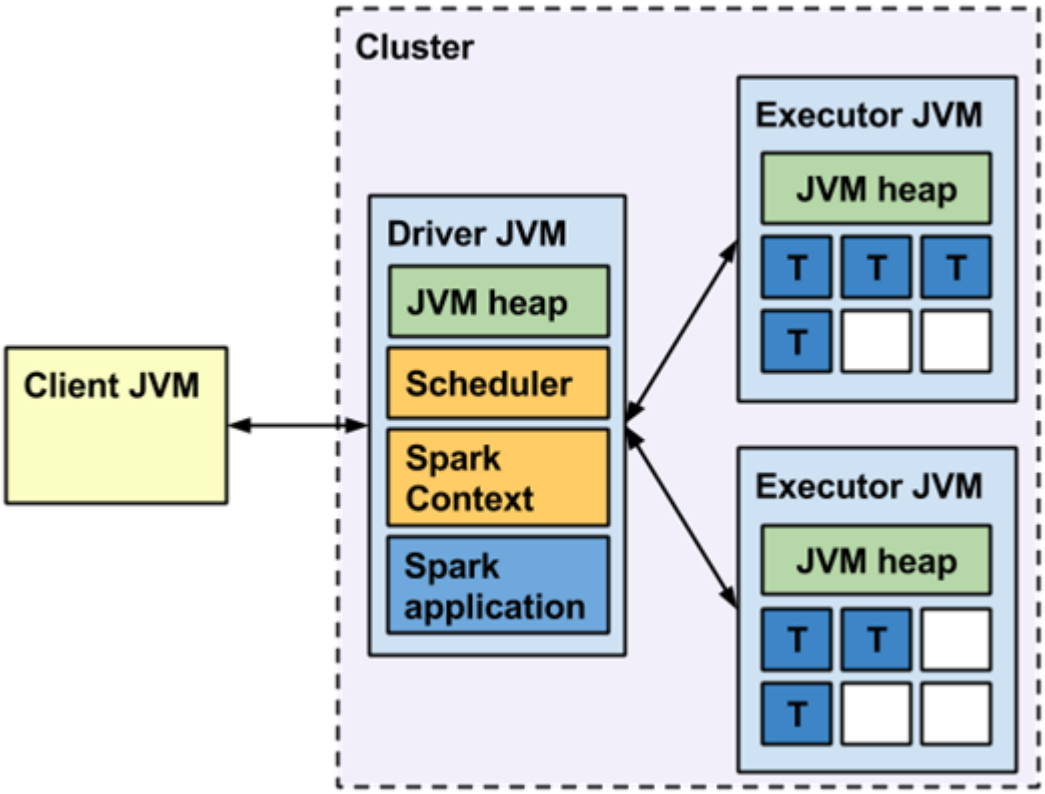
1. To allow Java programs to run on any device or operating system (known as the *"Write once, run anywhere"* principle)
2. To manage and optimize program memory

JVM view of the Spark Cluster: *Drivers, Executors, Slots & Tasks*

The Spark runtime architecture leverages JVMs:



And a slightly more detailed view:



Elements of a Spark application are in blue boxes and an application's tasks running inside task slots are labeled with a "T". Unoccupied task slots are in white boxes.

Responsibilities of the client process component

The **client** process starts the **driver** program. For example, the client process can be a `spark-submit` script for running applications, a spark-shell script, or a custom application using Spark API (like this Databricks **GUI** - **G**raphics **U**ser **I**nterface). The client process prepares the classpath and all configuration options for the Spark application. It also passes application arguments, if any, to the application running inside the **driver**.

1A) Driver

The **driver** orchestrates and monitors execution of a Spark application. There's always **one driver per Spark application**. You can think of the driver as a wrapper around the application.

The **driver** process runs our `main()` function, sits on a node in the cluster, and is responsible for:

1. Maintaining information about the Spark Application
2. Responding to a user's program or input
3. Requesting memory and CPU resources from cluster managers
4. Breaking application logic into stages and tasks
5. Sending tasks to executors
6. Collecting the results from the executors

The driver process is absolutely essential - it's the heart of a Spark Application and maintains all relevant information during the lifetime of the application.

- The **Driver** is the JVM in which our application runs.
- The secret to Spark's awesome performance is parallelism:
  - Scaling **vertically** (*i.e. making a single computer more powerful by adding physical hardware*) is limited to a finite amount of RAM, Threads and CPU speeds, due to the nature of motherboards having limited physical slots in Data Centers/Desktops.
  - Scaling **horizontally** (*i.e. throwing more identical machines into the Cluster*) means we can simply add new "nodes" to the cluster almost endlessly, because a Data Center can theoretically have an interconnected number of ~infinite machines
- We parallelize at two levels:
  - The first level of parallelization is the **Executor** - a JVM running on a node, typically, **one executor instance per node**.
  - The second level of parallelization is the **Slot** - the number of which is **determined by the number of cores and CPUs of each node/executor**.

1B) Executor

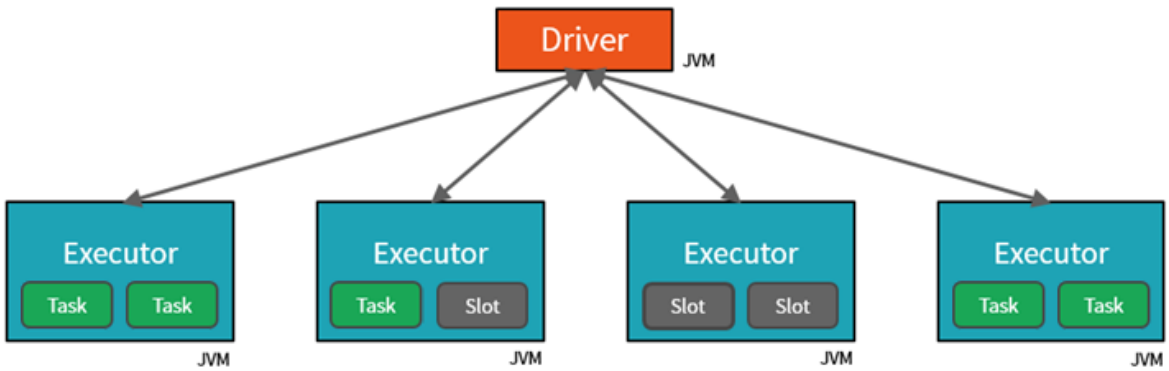
The **executors** are responsible for actually executing the work that the **driver** assigns them. This means, each executor is responsible for only two things:

1. Executing code assigned to it by the driver
2. Reporting the state of the computation, on that executor, back to the driver node

1C) Cores/Slots/Threads

- Each **Executor** has a number of **Slots** to which parallelized **Tasks** can be assigned to it by the **Driver**.
  - So for example:
    - If we have **3** identical home desktops (*nodes*) hooked up together in a LAN (like through your home router), each with i7 processors (**8** cores), then that's a **3** node Cluster:
      - **1** Driver node
      - **2** Executor nodes
    - The **8 cores per Executor node** means **8 Slots**, meaning the driver can assign each executor up to **8 Tasks**
      - The idea is, an i7 CPU Core is manufactured by Intel such that it is capable of executing it's own Task independent of the other Cores, so **8 Cores = 8 Slots = 8 Tasks in parellel**

For example: the diagram below is showing 2 Core Executor nodes:



- The JVM is naturally multithreaded, but a single JVM, such as our **Driver**, has a finite upper limit.
- By creating **Tasks**, the **Driver** can assign units of work to **Slots** on each **Executor** for parallel execution.
- Additionally, the **Driver** must also decide how to partition the data so that it can be distributed for parallel processing (see below).
  - Consequently, the **Driver** is assigning a **Partition** of data to each task - in this way each **Task** knows which piece of data it is to process.
  - Once started, each **Task** will fetch from the original data source (e.g. An Azure Storage Account) the **Partition** of data assigned to it.

Note relating to Tasks, Slots and Cores

You can set the number of task slots to a value two or three times (**i.e. to a multiple of**) the number of CPU cores. Although these task slots are often referred to as CPU cores in Spark, they're implemented as **threads** that work on a **physical core's thread** and don't need to correspond to the number of physical CPU cores on the machine (since different CPU manufacturer's can architect multi-threaded chips differently).

In other words:

- All processors of today have multiple cores (e.g. 1 CPU = 8 Cores)
- Most processors of today are multi-threaded (e.g. 1 Core = 2 Threads, 8 cores = 16 Threads)
- A Spark **Task** runs on a **Slot**. **1 Thread** is capable of doing **1 Task** at a time. To make use of all our threads on the CPU, we cleverly assign the **number of Slots** to correspond to a **multiple of the number of Cores** (which translates to multiple Threads).
  - *By doing this*, after the Driver breaks down a given command (`DO STUFF FROM massive_table`) into **Tasks** and **Partitions**, which are tailor-made to fit our particular Cluster Configuration (say *4 nodes - 1 driver and 3 executors, 8 cores per node, 2 threads per core*). By using our Clusters at maximum efficiency like this (utilizing all available threads), we can get our massive command executed as fast as possible (given our Cluster in this case, *3*8*2 Threads --> **48** Tasks, **48** Partitions* - i.e. **1** Partition per Task)
  - *Say we don't do this*, even with a 100 executor cluster, the entire burden would go to 1 executor, and the other 99 will be sitting idle - i.e. slow execution.
  - *Or say, we instead foolishly assign **49** Tasks and **49** Partitions*, the first pass would execute **48** Tasks in parallel across the executors cores (say in **10 minutes**), then that **1** remaining Task in the next pass will execute on **1** core for another **10 minutes**, while the rest of our **47** cores are sitting idle - meaning the whole job will take double the time at **20 minutes**. This is obviously an inefficient use of our available resources, and could rather be fixed by setting the number of tasks/partitions to a multiple of the number of cores we have (in this setup - 48, 96 etc).

1D) Partitions

DataFrames

A DataFrame is the most common Structured API and simply represents a **table of data with *rows* and *columns***. The list of columns and the types in those columns is called **the schema**.

```
cached.printSchema()
```

```
root
 |-- author: string (nullable = true)
 |-- avg_score: long (nullable = true)
 |-- comment: string (nullable = true)
 |-- controversiality: long (nullable = true)
 |-- created: long (nullable = true)
 |-- distinguished: string (nullable = true)
 |-- downs: long (nullable = true)
 |-- name: string (nullable = true)
 |-- subr: string (nullable = true)
 |-- ups: long (nullable = true)
 |-- url: string (nullable = true)
```

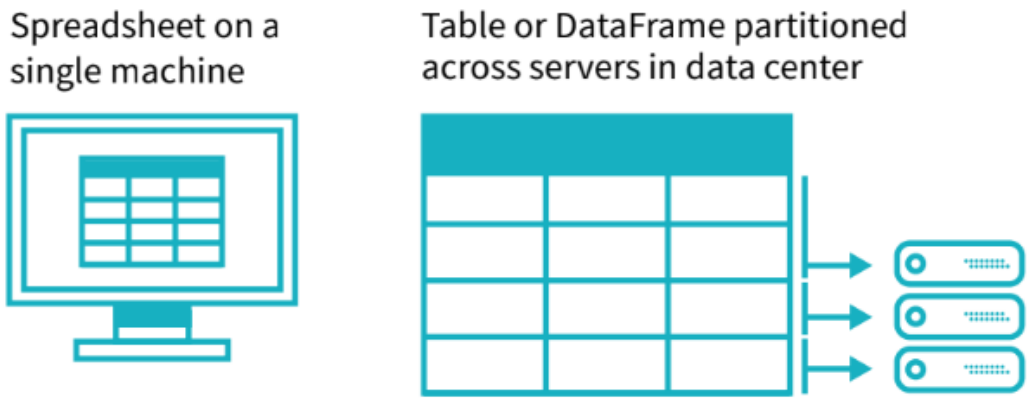
A simple analogy would be a spreadsheet with named columns. The **fundamental difference** is that while a spreadsheet sits on **one computer** in one specific location (e.g. *C:\Users\raki.rahman\Documents\MyFile.csv*), a Spark DataFrame can span **thousands of computers**.

The reason for putting the data on more than one computer is intuitive:

- Either ***the data is too large to fit on one machine*** or it would simply ***take too long to perform that computation on one machine***.

Data Partitions

In order to allow every executor to perform work in parallel, Spark breaks up the data into *chunks*, called **partitions**.



A **partition** is a collection of rows that sit on one physical machine in our cluster. A DataFrame's partitions represent how the data is physically distributed across your cluster of machines during execution:

- If you have *one* partition, Spark will only have a parallelism of *one*, even if you have thousands of executors.
- If you have *many* partitions, but only *one* executor, Spark will still only have a parallelism of *one* because there is only one computation resource.

An important thing to note is that with DataFrames, we do not (for the most part) manipulate partitions manually (on an individual basis). We simply specify high level transformations of data in the physical partitions and Spark determines how this work will actually execute on the cluster.

The key points to understand are that:

- Spark employs a **Cluster Manager** that is responsible for provisioning nodes in our cluster.
    - Databricks provides a robust, high-performing **Cluster Manager** as part of its overall offerings.
- In each of these scenarios, the **Driver** is running on one node, with each **Executors** running on N different nodes.

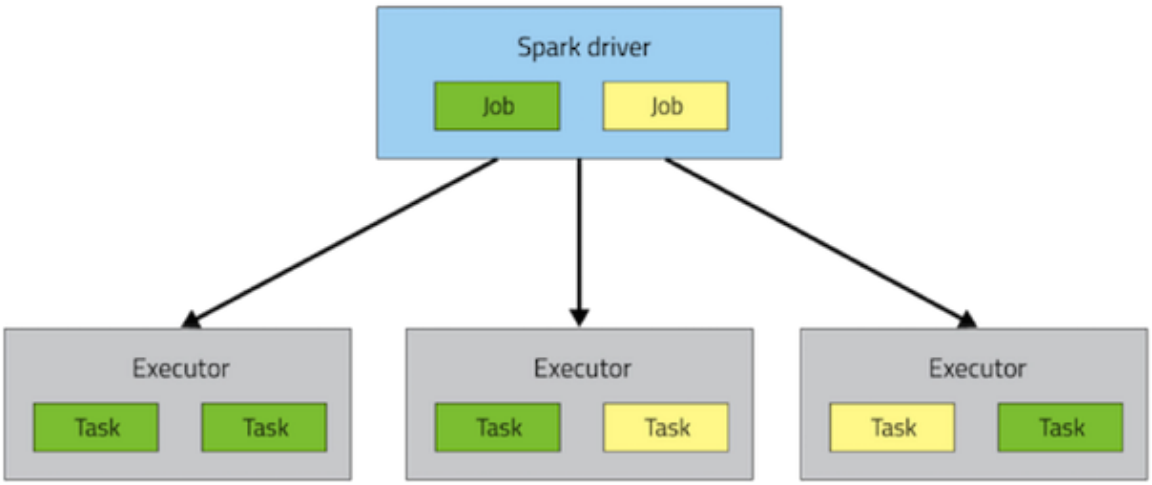
    - Databricks abstracts away the Cluster Management aspects for us (which is a massive pain)
- From a developer's and student's perspective the primary focus is on:
    - The number of **Partitions** the data is divided into
    - The number of **Slots** available for parallel execution
    - How many **Jobs** are being triggered?
    - And lastly the **Stages** those jobs are divided into

2. Spark Execution

Candidates are expected to be familiar with Spark's execution model and the breakdown between the different elements:

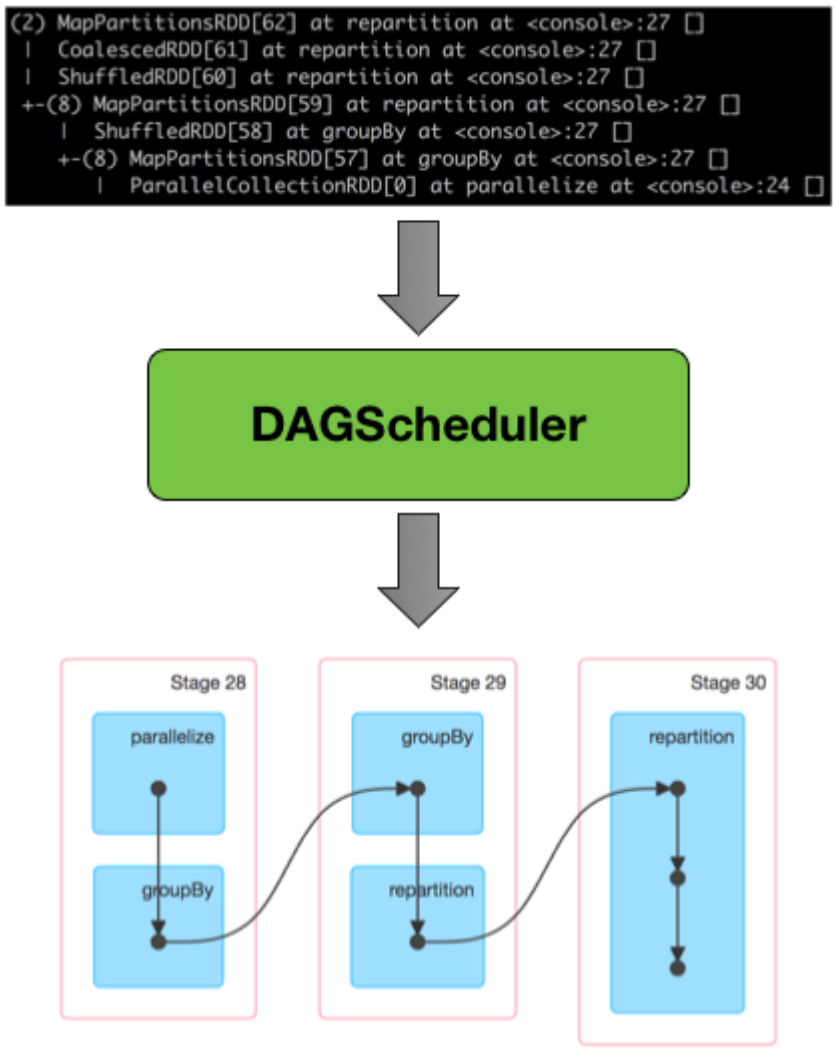
In Spark, the highest-level unit of computation is an **application**. A Spark application can be used for a single batch job, an interactive session with multiple jobs, or a long-lived server continually satisfying requests.

Spark **application execution**, alongside **drivers** and **executors**, also involves runtime concepts such as **tasks**, **jobs**, and **stages**. Invoking an **action** inside a Spark application triggers the launch of a **job** to fulfill it. Spark examines the dataset on which that action depends and formulates an **execution plan**. The **execution plan** assembles the dataset transformations into **stages**. A **stage** is a collection of tasks that run the same code, each on a different subset of the data.



Overview of DAGSchedular

DAGScheduler is the scheduling layer of Apache Spark that implements stage-oriented scheduling. It transforms a *logical* execution plan to a *physical* execution plan (using stages).



After an **action** (see below) has been called, `SparkContext` hands over a logical plan to **DAGScheduler** that it in turn translates to a set of **stages** that are submitted as a set of **tasks** for execution.

The fundamental concepts of **DAGScheduler** are **jobs** and **stages** that it tracks through internal registries and counters.

2A) Jobs

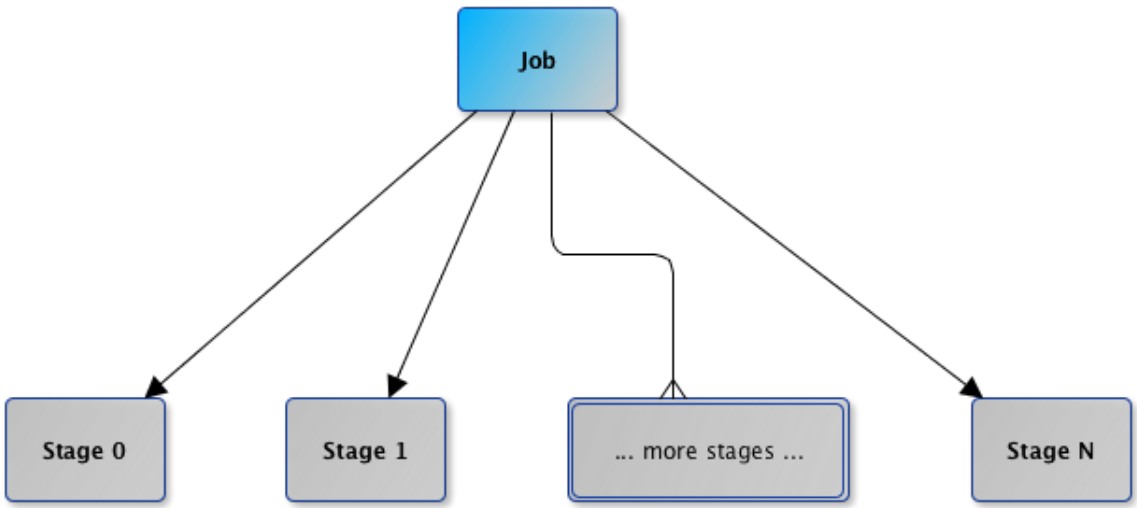
A **Job** is a sequence of **stages**, triggered by an **action** such as `count()`, `collect()`, `read()` or `write()`.

- Each parallelized action is referred to as a **Job**.
- The results of each **Job** (parallelized/distributed action) is returned to the **Driver** from the **Executor**.
- Depending on the work required, multiple **Jobs** will be required.

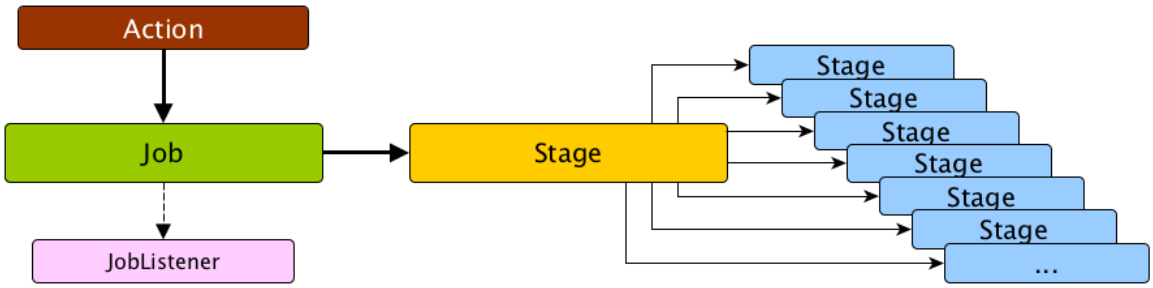
2B) Stages

*Each **job** that gets divided into smaller **sets of tasks** is a **stage**.*

A **Stage** is a sequence of **Tasks** that can all be run together - i.e. in parallel - without a **shuffle**. For example: using `.read` to read a file from disk, then runnning `.filter` can be done without a **shuffle**, so it can fit in a single **stage**. The number of **Tasks** in a **Stage** also depends upon the number of **Partitions** your datasets have.



- Each **Job** is broken down into **Stages**.
- This would be analogous to building *a house* (the job) - attempting to do any of these steps out of order doesn't make sense:
  1. Lay the foundation
  2. Erect the walls
  3. Add the rooms



In other words:

- A **stage** is a step in a physical execution plan - a physical unit of the execution plan
- A **stage** is a set of parallel **tasks - one task per partition - the blue boxes on the right of the diagram above** (of an RDD that computes *partial* results of a function executed as part of a Spark job).
- A Spark **job** is a computation with that computation sliced into **stages**
- A **stage** is uniquely identified by `id`. When a **stage** is created, **DAGScheduler** increments internal counter `nextStageId` to track the number of stage submissions.
- Each **stage** contains a sequence of **narrow transformations** (see below) that can be completed without **shuffling** the entire data set, separated at *shuffle boundaries*, i.e. where shuffle occurs. **Stages** are thus a result of breaking the RDD at shuffle boundaries.

An example of Stages - *Inefficient*

- When we shuffle data, it creates what is known as a *stage boundary*.
- Stage boundaries represent a **process bottleneck**.

Take for example the following transformations:

Step	Transformation
1	`Read`
2	`Select`
3	`Filter`
4	`GroupBy`
5	`Select`
6	`Filter`
7	`Write`

Spark will break this one job into two stages (steps *1-4b* and steps *4c-8*):

Stage #1

Step	Transformation
1	`Read`
2	`Select`
3	`Filter`
4a	`GroupBy` ***1***/2
4b	`shuffle write`

Stage #2

Step	Transformation
4c	`shuffle read`
4d	`GroupBy` ***2***/2
5	`Select`
6	`Filter`
7	`Write`

In **Stage #1**, Spark will create a **pipeline of transformations** in which the data is read into RAM (Step #1), and then perform steps #2, #3, #4a & #4b

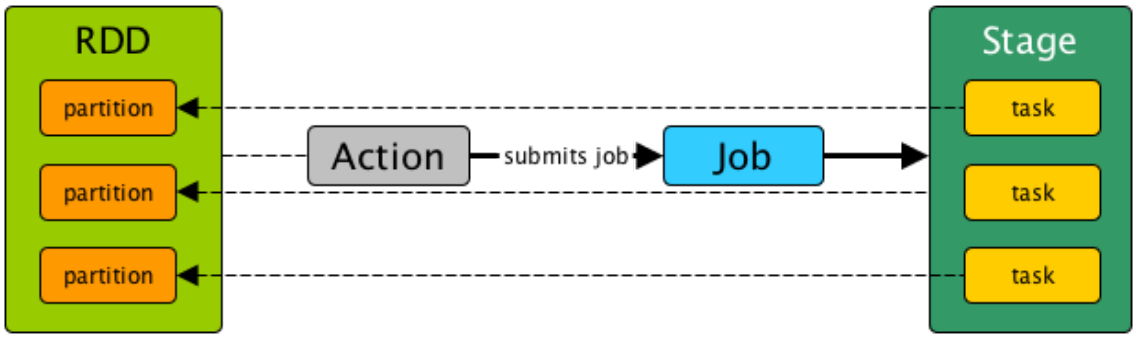
All partitions must complete **Stage #1** before continuing to **Stage #2**

- It's not possible to `Group` all records across all partitions until every task is completed.
- This is the point at which all the tasks (across the executor slots) must synchronize.
- This creates our **bottleneck**.
- Besides the bottleneck, this is also a significant **performance hit**: *disk IO, network IO and more disk IO*.

Once the data is shuffled, we can resume execution.

For **Stage #2**, Spark will again create a pipeline of transformations in which the shuffle data is read into RAM (Step #4c) and then perform transformations #4d, #5, #6 and finally the write action, step #7.

2C) Tasks



A **task** is a unit of work that is sent to the executor. Each **stage** has some tasks, one **task per partition**. The same task is done over different partitions of the RDD.

In the *example of Stages* above, each **Step** is a **Task**.

An example of Stages - *Efficient*

Working Backwards

From the developer's perspective, we start with a read and conclude (in this case) with a write

Step	Transformation
1	`Read`
2	`Select`
3	`Filter`
4	`GroupBy`
5	`Select`
6	`Filter`
7	`Write`

However, Spark starts backwards with the **action** (`write(..)` in this case).

Next, it asks the question, **what do I need to do first**?

It then proceeds to determine which transformation precedes this step until it identifies the first transformation.

Step	Transformation	Dependencies
7	`Write`	Depends on #6
6	`Filter`	Depends on #5
5	`Select`	Depends on #4
4	`GroupBy`	Depends on #3
3	`Filter`	Depends on #2
2	`Select`	Depends on #1
1	`Read`	First

This would be equivalent to understanding your own lineage.

- You don't ask if you are related to *Genghis Khan* and then work through the ancestry of all his children (5% of all people in Asia).
  - You start with your mother.
  - Then your grandmother
  - Then your great-grandmother
  - ... and so on
  - Until you discover you are actually related to *Catherine Parr, the last queen of Henry the VIII*.

Why Work Backwards?

Question: So what is the benefit of working backward through your action's lineage?
Answer: It allows Spark to determine if it is necessary to execute every transformation.

Take another look at our example:

- Say we've executed this once already
- On the first execution, **Step #4** resulted in a shuffle
- Those shuffle files are on the various **executors** already (src & dst)
- Because the **transformations** (or DataFrames) are **immutable**, no aspect of our lineage can change (meaning that DataFrame is sitting on a chunk of the executor's RAM from the last time it was calculated, ready to be referenced again).
- That means the results of our *previous* **shuffle** (if still available) can be reused.

Step	Transformation	
7	`Write`	Depends on #6
6	`Filter`	Depends on #5
5	`Select`	Depends on #4
4	`GroupBy`	<<< shuffle
3	`Filter`	*don't care*
2	`Select`	*don't care*
1	`Read`	*don't care*

In this case, what we end up executing is only the operations from **Stage #2**.

This saves us the initial network read and all the transformations in **Stage #1**

Step	Transformation	
1	`Read`	*skipped*
2	`Select`	*skipped*
3	`Filter`	*skipped*
4a	`GroupBy` ***1****/2*	*skipped*
4b	`shuffle write`	*skipped*
4c	`shuffle read`	-
4d	`GroupBy` ***2****/2*	-
5	`Select`	-
6	`Filter`	-
7	`Write`	-

Summary: Jobs, Stages, Tasks

Jobs

- Highest element of Spark's execution hierarchy.
- Each Spark job corresponds to one **Action**

Stages

- As mentioned above, a job is defined by calling an action.
- The action may include several **transformations**, which breaks down **jobs** into **stages**.

- Several **transformations** with *narrow* dependencies can be grouped into one stage
- It is possible to execute **stages** in *parallel* if they are used to compute different RDDs
- *Wide* **transformations** that are needed to compute one RDD have to be computed in sequence
- One **stage** can be computed without moving data across the partitions
- Within one **stage**, the **tasks** are the unit of work done for each partition of the data

Tasks

- A **stage** consists of **tasks**
- The **task** is the smallest unit in the execution hierarchy
- Each **task** can represent one local computation
- One **task** cannot be executed on more than one **executor**
- However, each **executor** has a dynamically allocated number of **slots** for running **tasks**
- The number of **tasks** per **stage** corresponds to the number of partitions in the output RDD of that **stage**

3. Spark Concepts

Candidates are expected to be familiar with the following concepts:

3A) Caching

The reuse of shuffle files (aka our temp files) is just one example of Spark optimizing queries anywhere it can.

We cannot assume this will be available to us.

Shuffle files are by *definition* **temporary files** and will eventually be removed.

However, we can cache data to **explicitly** to accomplish the same thing that happens inadvertently (i.e. we get *lucky*) with shuffle files.

In this case, the lineage plays the same role. Take for example:

Step	Transformation	
7	`Write`	Depends on #6
6	`Filter`	Depends on #5
5	`Select`	<<< cache
4	`GroupBy`	<<< shuffle files
3	`Filter`	?
2	`Select`	?
1	`Read`	?

In this case we **explicitly asked Spark to cache** the DataFrame resulting from the `select(..)` in Step 5 (after the shuffle across the network due to `GroupBy`.

As a result, we never even get to the part of the lineage that involves the shuffle, let alone **Stage #1** (i.e. we skip the whole thing, making our job execute faster).

Instead, we pick up with the cache and resume execution from there:

Step	Transformation	
1	`Read`	*skipped*
2	`Select`	*skipped*
3	`Filter`	*skipped*
4a	`GroupBy` ***1/2***	*skipped*
4b	`shuffle write`	*skipped*
4c	`shuffle read`	*skipped*
4d	`GroupBy` ***2/2***	*skipped*
5a	`cache read`	-
5b	`Select`	-
6	`Filter`	-
7	`Write`	-

3B) Shuffling

A Shuffle refers to an operation where data is *re-partitioned* across a **Cluster** - i.e. when data needs to move between executors.

`join` and any operation that ends with `ByKey` will trigger a **Shuffle**. It is a costly operation because a lot of data can be sent via the network.

For example, to group by color, it will serve us best if...

- All the reds are in one partitions
- All the blues are in a second partition
- All the greens are in a third

From there we can easily sum/count/average all of the reds, blues, and greens.

To carry out the shuffle operation Spark needs to

- Convert the data to the UnsafeRow (if it isn't already), commonly refered to as **Tungsten Binary Format**.
  - **Tungsten** is a new Spark SQL component that provides more efficient Spark operations by working directly at the byte level.
  - Includes specialized in-memory data structures tuned for the type of operations required by Spark
  - Improved code generation, and a specialized wire protocol.
- Write that data to disk on the **local node** - at this point the slot is free for the next task.
- Send that data across the **network** to another **executor**
  - **Driver** decides which **executor** gets which partition of the data.
  - Then the **executor** pulls the data it needs from the other executor's shuffle files.
- Copy the data back into **RAM** on the new executor
  - The concept, if not the action, is just like the initial read "every" `DataFrame` starts with.
  - The main difference being it's the 2nd+ stage.

This amounts to a free cache from what is effectively temp files.

Note: *Some actions induce in a shuffle.*
Good examples would include the operations `count()` *and* `reduce(..)`*.*

3C) Partitioning

A **Partition** is a logical chunk of your **DataFrame**

Data is split into **Partitions** so that each **Executor** can operate on a single part, enabling **parallelization**.

It can be processed by a **single Executor core/thread**.

For example: If you have **4** data partitions and you have **4** executor cores/threads, you can process everything in parallel, in a single pass.

3D) DataFrame Transformations vs. Actions vs. Operations

Spark allows two distinct kinds of operations by the user: **transformations** and **actions**.

*Transformations are **LAZY**, Actions are **EAGER***

Transformations - Overview

Transformations are operations that will not be completed at the time you write and execute the code in a cell (they're **lazy**) - they will only get executed once you have called an **action**. An example of a transformation might be to convert an `integer` into a `float` or to `filter` a set of values: i.e. they can be procrastinated and don't have to be done *right now* - but later after we have a full view of the task at hand.

Here's an analogy:

- Let's say you're cleaning your closet, and want to donate clothes that don't fit (there's a lot of these starting from childhood days), and sort out and store the rest by color before storing in your closet.
- If you're **inefficient**, you could sort out all the clothes by color (let's say that takes *60 minutes*), then from there pick the ones that fit (*5 minutes*), and then take the rest and put it into one big plastic bag for donation (where all that sorting effort you did went to waste because it's all jumbled up in the same plastic bag now anyway)
- If you're **efficient**, you'd first pick out clothes that fit very quickly (*5 minutes*), then sort those into colors (*10 minutes*), and then take the rest and put it into one big plastic bag for donation (where there's no wasted effort)

In other words, by evaluating the full view of the **job** at hand, and by being **lazy** (not in the traditional sense - but in the smart way by **not eagerly** sorting everything by color for no reason), you were able to achieve the same goal in *15 minutes* vs *65 minutes* (clothes that fit are sorted by color in the closet, clothes that don' fit are in plastic bag).

Actions - Overview

Actions are commands that are computed by Spark right at the time of their execution (they're **eager**). They consist of running all of the previous transformations in order to get back an actual result. An **action** is composed of one or more **jobs** which consists of **tasks** that will be executed by the **executor slots** in parallel - i.e. a **stage** - where possible.

Here are some simple examples of transformations and actions.


transformations and actions

Spark `pipelines` a computation as we can see in the image below. This means that certain computations can all be performed at once (like a `map` and a `filter`) rather than having to do one operation for all pieces of data, and then the following operation.


transformations and actions

Why is Laziness So Important?

It has a number of benefits:

- Not forced to load all data at step #1
  - Technically impossible with **REALLY** large datasets.
- Easier to parallelize operations
  - *N* different transformations can be processed on a single data element, on a single thread, on a single machine.
- Most importantly, it allows the framework to automatically apply various optimizations

Actions

Transformations always return a `DataFrame`.

In contrast, Actions either return a *result* or *write to disk*. For example:

- The number of records in the case of `count()`
- An array of objects in the case of `collect()` or `take(n)`

We've seen a good number of the actions - most of them are listed below.

For the complete list, one needs to review the API docs.

Method	Return	Description
`collect()`	Collection	Returns an array that contains all of Rows in this Dataset.
`count()`	Long	Returns the number of rows in the Dataset.
`first()`	Row	Returns the first row.
`foreach(f)`	-	Applies a function f to all rows.
`foreachPartition(f)`	-	Applies a function f to each partition of this Dataset.
`head()`	Row	Returns the first row.
`reduce(f)`	Row	Reduces the elements of this Dataset using the specified binary function.
`show(..)`	-	Displays the top 20 rows of Dataset in a tabular form.
`take(n)`	Collection	Returns the first n rows in the Dataset.
`toLocalIterator()`	Iterator	Return an iterator that contains all of Rows in this Dataset.

***Note:** The databricks command `display(..)` is not included here because it's not part of the Spark API, even though it ultimately calls an action.*

Transformations

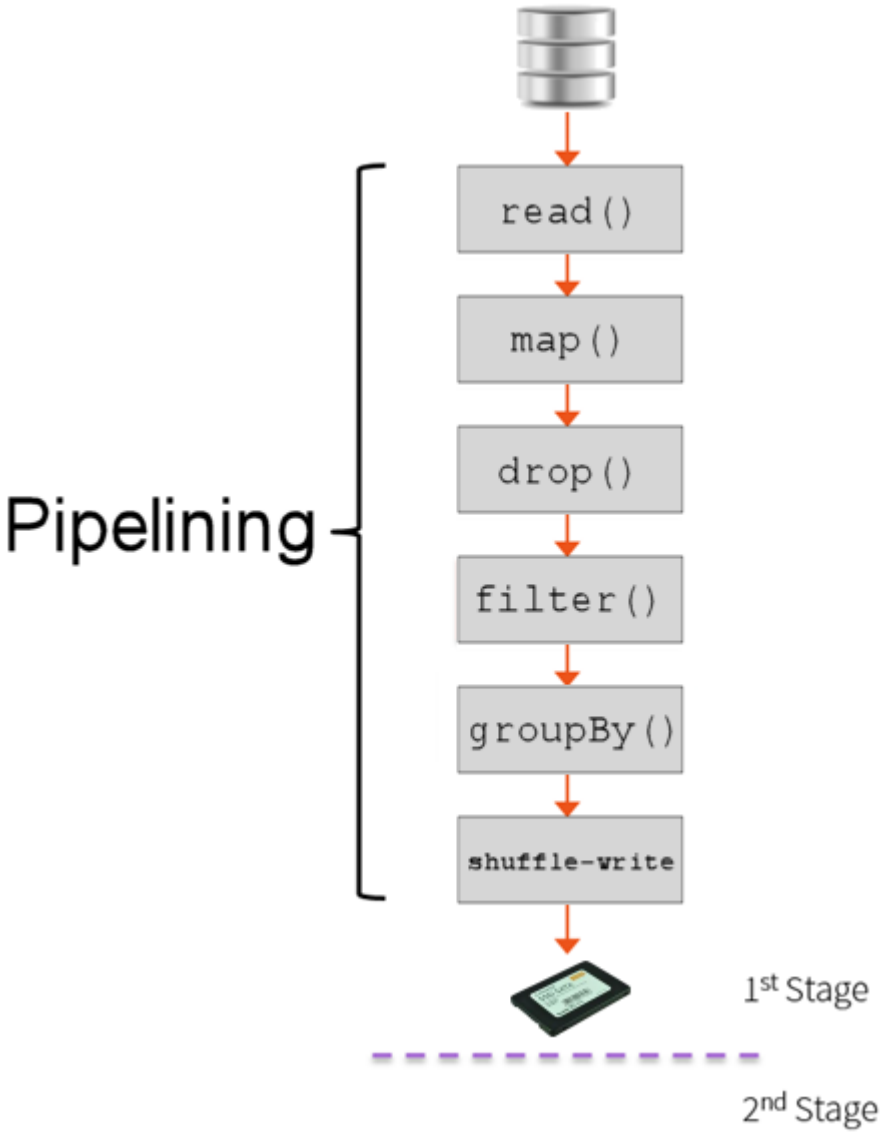
Transformations have the following key characteristics:

- They eventually return another `DataFrame` .
- DataFrames are **immutable** - that is each instance of a `DataFrame` cannot be altered once it's instantiated.
  - This means other optimizations are possible - such as the use of shuffle files (see below)
- Are classified as either a **Wide** or **Narrow** transformation

Note: The list of transformations varies significantly between each language - because Java & Scala are *strictly* typed languages compared Python & R which are *loosely* typed.

Pipelining Operations

- Pipelining is the idea of executing as many **operations** as possible on a single partition of data.
- Once a single partition of data is read into RAM, Spark will combine as many **narrow operations** as it can into a single **task**
- **Wide operations** force a shuffle, conclude, a stage and end a pipeline.



3E) Wide vs. Narrow Transformations

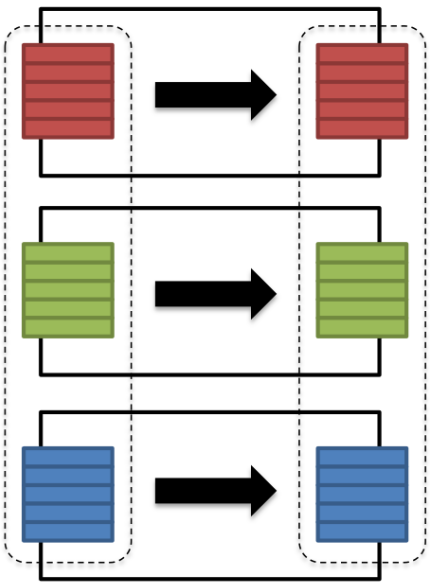
Wide vs. Narrow Transformations

Regardless of language, transformations break down into two broad categories: **wide** and **narrow**.

Narrow Transformations: The data required to compute the records in a single partition reside in at most one partition of the parent RDD.

Examples include:

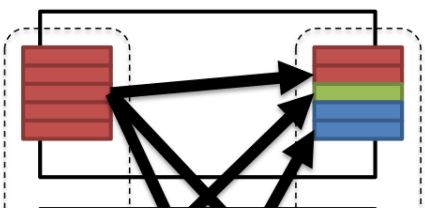
- `filter(..)`
- `drop(..)`
- `coalesce()`

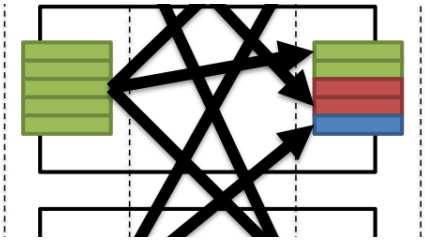


Wide Transformations: The data required to compute the records in a single partition may reside in many partitions of the parent RDD.

Examples include:

- `distinct()`
- `groupBy(..).sum()`
- `repartition(n)`





3F) High-level Cluster Configuration

Spark cluster types

Spark can run in **local mode** (on your laptop) and inside **Spark standalone**, **YARN**, and **Mesos** clusters. Although Spark runs on all of them, one might be more applicable for your environment and use cases. In this section, you'll find the pros and cons of each cluster type.

Spark local modes

Spark local mode and Spark local cluster mode are special cases of a Spark standalone cluster running on a single machine. Because these cluster types are easy to set up and use, they're convenient for quick tests, but they shouldn't be used in a production environment.

Furthermore, in these local modes, the workload isn't distributed, and it creates the resource restrictions of a single machine and suboptimal performance. True high availability isn't possible on a single machine, either.

Spark standalone cluster

A Spark standalone cluster is a Spark-specific cluster. Because a standalone cluster is built specifically for Spark applications, it doesn't support communication with an HDFS secured with Kerberos authentication protocol. If you need that kind of security, use YARN for running Spark.

YARN cluster

YARN is Hadoop's resource manager and execution system. It's also known as *MapReduce 2* because it superseded the *MapReduce* engine in *Hadoop 1* that supported only MapReduce jobs.

Running Spark on YARN has several advantages:

- Many organizations already have YARN clusters of a significant size, along with the technical know-how, tools, and procedures for managing and monitoring them.
- Furthermore, YARN lets you run different types of Java applications, not only Spark, and you can mix legacy Hadoop and Spark applications with ease.
- YARN also provides methods for isolating and prioritizing applications among users and organizations, a functionality the standalone cluster doesn't have.
- It's the only cluster type that supports **Kerberos-secured HDFS**.
- Another advantage of YARN over the standalone cluster is that you don't have to install Spark on every node in the cluster.

Mesos cluster

Mesos is a scalable and fault-tolerant "distributed systems kernel" written in C++. Running Spark in a Mesos cluster also has its advantages. Unlike YARN, Mesos also supports C++ and Python applications, and unlike YARN and a standalone Spark cluster that only schedules memory, Mesos provides scheduling of other types of resources (for example, CPU, disk space and ports), although these additional resources aren't used by Spark currently. Mesos has some additional options for job scheduling that other cluster types don't have (for example, fine-grained mode).

And, Mesos is a "scheduler of scheduler frameworks" because of its two-level scheduling architecture. The jury's still out on which is better: YARN or Mesos; but now, with the Myriad project (http://myriad.incubator.apache.org/), you can run YARN on top of Mesos to solve the dilemma.

DataFrames API

Candidates are expected to have a command of the following APIs.

4. SparkContext

Overview

Once the **driver** is started, it configures an instance of `SparkContext`. Your Spark context is already preconfigured and available as the variable `sc`. When running a standalone Spark application by submitting a jar file, or by using Spark API from another program, your Spark application starts and configures the Spark context (i.e. Databricks).

Note: There is usually one Spark context per JVM. Although the configuration option `spark.driver.allowMultipleContexts` exists, it's misleading because usage of multiple Spark contexts is discouraged. This option is used only for Spark internal tests and we recommend you don't use that option in your user programs. If you do, you may get unexpected results while running more than one Spark context in a single JVM.

A Spark context comes with many useful methods for creating DataFrames, loading data (e.g. `spark.read.format("csv")`), and is the main interface for accessing Spark runtime.

4A) `SparkContext` to control basic configuration settings such as `spark.sql.shuffle.partitions`.

Definitions

`spark.sql.shuffle.partitions`: Configures the number of partitions to use when shuffling data for joins or aggregations.
`spark.executor.memory`: Amount of memory to use per executor process, in the same format as JVM memory strings with a size unit suffix ("k", "m", "g" or "t") (e.g. 512m, 2g).
`spark.default.parallelism`: Default number of partitions in RDDs returned by transformations like `join`, `reduceByKey`, and `parallelize` when not set by user. Note that this is **ignored for DataFrames**, and we can use `df.repartition(numOfPartitions)` instead.

Let's set the value of `spark.sql.shuffle.partitions` and `spark.executor.memory` using PySpark and SQL syntax.

```
# Print the default values of shuffle partition and the executor memory
print(spark.conf.get("spark.sql.shuffle.partitions"), ",",
spark.conf.get("spark.executor.memory"))
```

```
200 , 7284m
```

```
# Set the number of shuffle partitions to 6
spark.conf.set("spark.sql.shuffle.partitions", 6)
# Set the memory of executors to 2 GB
spark.conf.set("spark.executor.memory", "2g")
# Print the values of the shuffle partition and the executor memory
print(spark.conf.get("spark.sql.shuffle.partitions"), ",",
spark.conf.get("spark.executor.memory"))
```

```
6 , 2g
```

```
%sql
SET spark.sql.shuffle.partitions = 200;
```

key
spark.sql.shuffle.partitions

```
%sql
SET spark.executor.memory = 7284m;
```

key
spark.executor.memory

5. SparkSession

Candidates are expected to know how to:

5A) Create a *DataFrame/Dataset* from a

collection (e.g. `list` or `set`)

```
# import relevant modules
from pyspark.sql import *
from pyspark.sql.types import *
from pyspark.sql.functions import *
from pyspark import *
from pyspark import StorageLevel
import sys
```

Example: Create DataFrame from `list` with `DataType` specified

```
list_df = spark.createDataFrame([1, 2, 3, 4], IntegerType())
display(list_df)
```

value
1
2
3
4

Example: Create DataFrame from `Row`

class `pyspark.sql.Row`

A row in `DataFrame`. The fields in it can be accessed:

- like attributes (`row.key`)
- like dictionary values (`row[key]`)

In this scenario we have two tables to be joined `employee` and `department`. Both tables contains only a few records, but we need to join them to get to know the department of each employee. So, we join them using Spark DataFrames like this:

```
# Create Example Data - Departments and Employees

# Create the Employees
Employee = Row("name") # Define the Row `Employee' with one column/key
employee1 = Employee('Bob') # Define against the Row 'Employee'
employee2 = Employee('Sam') # Define against the Row 'Employee'

# Create the Departments
Department = Row("name", "department") # Define the Row `Department' with
two columns/keys
department1 = Department('Bob', 'Accounts') # Define against the Row
'Department'
department2 = Department('Alice', 'Sales') # Define against the Row
'Department'
department3 = Department('Sam', 'HR') # Define against the Row 'Department'

# Create DataFrames from rows
employeeDF = spark.createDataFrame([employee1, employee2])
departmentDF = spark.createDataFrame([department1, department2,
department3])

# Join employeeDF to departmentDF on "name"
display(employeeDF.join(departmentDF, "name"))
```

name	de
Bob	Ac
Sam	HF

Example: Create DataFrame from `Row`, with Schema specified

`createDataFrame` (*data, schema=None, samplingRatio=None, verifySchema=True*)
Creates a `DataFrame` from an `RDD`, a `list` or a `pandas.DataFrame`. When `schema` is a list of column names, the type of each column will be inferred from `data`. When `schema` is `None`, it will try to infer the schema (column names and types) from data, which should be an RDD of `Row`, or `namedtuple`, or `dict`.

```
schema = StructType([
  StructField("letter", StringType(), True),
  StructField("position", IntegerType(), True)])

df = spark.createDataFrame([('A', 0),('B', 1),('C', 2)], schema)
display(df)
```

letter
A
B
C

Example: Create DataFrame from a list of `Rows`

```
# Create Example Data - Departments and Employees

# Create the Departments
Department = Row("id", "name")
department1 = Department('123456', 'Computer Science')
department2 = Department('789012', 'Mechanical Engineering')
department3 = Department('345678', 'Theater and Drama')
department4 = Department('901234', 'Indoor Recreation')
department5 = Department('000000', 'All Students')

# Create the Employees
Employee = Row("firstName", "lastName", "email", "salary")
employee1 = Employee('michael', 'armbrust', 'no-reply@berkeley.edu', 100000)
employee2 = Employee('xiangrui', 'meng', 'no-reply@stanford.edu', 120000)
employee3 = Employee('matei', None, 'no-reply@waterloo.edu', 140000)
employee4 = Employee(None, 'wendell', 'no-reply@berkeley.edu', 160000)
employee5 = Employee('michael', 'jackson', 'no-reply@neverla.nd', 80000)

# Create the DepartmentWithEmployees instances from Departments and
Employees
DepartmentWithEmployees = Row("department", "employees")
departmentWithEmployees1 = DepartmentWithEmployees(department1, [employee1,
employee2])
departmentWithEmployees2 = DepartmentWithEmployees(department2, [employee3,
employee4])
departmentWithEmployees3 = DepartmentWithEmployees(department3, [employee5,
employee4])
departmentWithEmployees4 = DepartmentWithEmployees(department4, [employee2,
employee3])
departmentWithEmployees5 = DepartmentWithEmployees(department5, [employee1,
employee2, employee3, employee4, employee5])

print(department1)
print(employee2)
print(departmentWithEmployees1.employees[0].email)

Row(id='123456', name='Computer Science')
Row(firstName='xiangrui', lastName='meng', email='no-reply@stanford.edu', sa
lary=120000)
no-reply@berkeley.edu

departmentsWithEmployeesSeq1 = [departmentWithEmployees1,
departmentWithEmployees2, departmentWithEmployees3,
departmentWithEmployees4, departmentWithEmployees5]
df1 = spark.createDataFrame(departmentsWithEmployeesSeq1)

display(df1)
```

department	employees
▸ {"id":"123456","name":"Computer Science"}	▸ [{"firstName":"michael","lastName":"armbrust","email":"no
▸ {"id":"789012","name":"Mechanical Engineering"}	▸ [{"firstName":"matei","lastName":null,"email":"no-reply@w
▸ {"id":"345678","name":"Theater and Drama"}	▸ [{"firstName":"michael","lastName":"jackson","email":"no-r
▸ {"id":"901234","name":"Indoor Recreation"}	▸ [{"firstName":"xiangrui","lastName":"meng","email":"no-re
▸ {"id":"000000","name":"All Students"}	▸ [{"firstName":"michael","lastName":"armbrust","email":"no {"firstName":"matei","lastName":null,"email":"no-reply@wa

5B) Create a *DataFrame* for a range of numbers

`range` (*start, end=None, step=1, numPartitions=None*)

Create a `DataFrame` with single `pyspark.sql.types.LongType` column named `id`, containing elements in a range from start to end (exclusive) with step value `step`.

Parameters

- **start** – the start value
- **end** – the end value (exclusive)
- **step** – the incremental step (default: 1)
- **numPartitions** – the number of partitions of the DataFrame

Returns:

- **`DataFrame`**

```
df = spark.range(1,8,2).toDF("number")
display(df)
```

number
1
3
5

7

```
df = spark.range(5).toDF("number")
display(df)
```

number
0
1
2
3
4

5C) Access the *DataFrameReaders*

DataFrameReader — Loading Data From External Data Sources

class `pyspark.sql.DataFrameReader` (spark)
Interface used to load a `DataFrame` from external storage systems (e.g. file systems, key-value stores, etc). Use `spark.read()` to access this.

Before we start with using `DataFrameReaders`, let's mount a storage container that will contain different files for us to use.

```
# Localize with Storage Account name and key
storageAccountName = "storage--acount--name"
storageAccountAccessKey = "storage--acount--key"

# Function to mount a storage container
def mountStorageContainer(storageAccount, storageAccountKey,
storageContainer, blobMountPoint):
  print("Mounting {0} to {1}:".format(storageContainer, blobMountPoint))
  # Attempt mount only if the storage container is not already mounted at
the mount point
  if not any(mount.mountPoint == blobMountPoint for mount in
dbutils.fs.mounts()):
    print("....Container is not mounted; Attempting mounting now..")
    mountStatus = dbutils.fs.mount(
                  source =
"wasbs://{0}@{1}.blob.core.windows.net/".format(storageContainer,
storageAccount),
                  mount_point = blobMountPoint,
                  extra_configs = {"fs.azure.account.key.
{0}.blob.core.windows.net".format(storageAccount): storageAccountKey})
    print("....Status of mount is: " + str(mountStatus))
  else:
    print("....Container is already mounted.")
    print() # Provide a blank line between mounts

# Mount "bronze" storage container
mountStorageContainer(storageAccountName,storageAccountAccessKey,"bronze","/
mnt/GoFast/bronze")

# Display directory
display(dbutils.fs.ls("/mnt/GoFast/bronze"))
```

path
dbfs:/mnt/GoFast/bronze/TestVectorOrcFile.testLzo.orc
dbfs:/mnt/GoFast/bronze/airbnb-sf-listings.csv
dbfs:/mnt/GoFast/bronze/auto-mpg.csv
dbfs:/mnt/GoFast/bronze/searose-env.csv
dbfs:/mnt/GoFast/bronze/tweets.txt
dbfs:/mnt/GoFast/bronze/wine.parquet/
dbfs:/mnt/GoFast/bronze/zips.json

We read in the `auto-mpg.csv` with a simple `spark.read.csv` command.

Note: This particular csv doesn't have a header specified.

```
df = spark.read.csv("/mnt/GoFast/bronze/auto-mpg.csv", header=False,
inferSchema=True)
display(df)
```

_c0	_c1	_c2	_c3	_c4
18	8	307	130	3504
15	8	350	165	3693
18	8	318	150	3436
16	8	304	150	3433
17	8	302	140	3449
15	8	429	198	4341
14	8	454	220	4354
14	8	440	215	4312
14	8	455	225	4425

5D) Register User Defined Functions (UDFs).

If we have a `function` that can use values from a row in the `dataframe` as input, then we can map it to the entire dataframe. The only difference is that with PySpark UDFs we have to specify the `output` data type.

We first **define the udf** below:

```
# Define UDF
def square(s):
  return s * s

# Register UDF to spark as 'squaredWithPython'
spark.udf.register("squaredWithPython", square, LongType())

Out[13]: <function __main__.square>
```

Note that we can call `squaredWithPython` immediately from SparkSQL. We first register a `temp` table called 'table':

```
# Register temptable with range of numbers
spark.range(0, 19, 3).toDF("num").createOrReplaceTempView("table")
```

```
%sql
SELECT num, squaredWithPython(num) as num_sq FROM table
```

num
0
3
6
9
12
15
18

But note that with `DataFrames`, this will not work until we define the UDF **explicitly** (on top of registering it above) with the respective return `DataType`. In other words, when we call `spark.udf.register` above, that registers it with spark SQL only, and for DataFrame it must be explicitly defined as an UDF.

```
# Convert temp table to DataFrame
tabledf = spark.table("table")

# Define UDF
squaredWithPython = udf(square, LongType())

display(tabledf.select("num", squaredWithPython("num").alias("num_sq")))
```

num
0
3
6
9
12
15
18

6. DataFrameReader

Candidates are expected to be familiar with the following architectural components and their relationship to each other.

6A) Read data for the “core” data formats (CSV, JSON, JDBC, ORC, Parquet, text and tables)

CSV

Let's read `airbnb-sf-listings.csv` from our mount, with the header specified, and infer schema on read.

```
csvdf = spark.read.csv("/mnt/GoFast/bronze/airbnb-sf-listings.csv",
header=True, inferSchema=True)
display(csvdf)
```

id	name	host_id	host_name	neighbourhood_group	neigh
958	Bright, Modern Garden Unit - 1BR/1B	1169	Holly	null	West
5858	Creative Sanctuary	8904	Philip And Tania	null	Berna
7918	A Friendly Room - UCSF/USF - San Francisco	21994	Aaron	null	Haigh
8014	Newly Remodeled room in big house WIFI market	22402	Jia	null	Outer

Showing the first 1000 rows.

JSON

Let's first view `zip.json` from our mount, and then load it into a DataFrame.

```
%fs head "dbfs:/mnt/GoFast/bronze/zips.json"
```

```
[Truncated to first 65536 bytes]
{ "_id" : "01001", "city" : "AGAWAM", "loc" : [ -72.622739, 42.070206 ], "po
p" : 15338, "state" : "MA" }
{ "_id" : "01002", "city" : "CUSHMAN", "loc" : [ -72.51564999999999, 42.3770
17 ], "pop" : 36963, "state" : "MA" }
{ "_id" : "01005", "city" : "BARRE", "loc" : [ -72.10835400000001, 42.409698
], "pop" : 4546, "state" : "MA" }
{ "_id" : "01007", "city" : "BELCHERTOWN", "loc" : [ -72.41095300000001, 42.
275103 ], "pop" : 10579, "state" : "MA" }
{ "_id" : "01008", "city" : "BLANDFORD", "loc" : [ -72.936114, 42.182949 ],
"pop" : 1240, "state" : "MA" }
{ "_id" : "01010", "city" : "BRIMFIELD", "loc" : [ -72.188455, 42.116543 ],
"pop" : 3706, "state" : "MA" }
{ "_id" : "01011", "city" : "CHESTER", "loc" : [ -72.988761, 42.279421 ], "p
op" : 1688, "state" : "MA" }
{ "_id" : "01012", "city" : "CHESTERFIELD", "loc" : [ -72.833309, 42.38167
], "pop" : 177, "state" : "MA" }
{ "_id" : "01013", "city" : "CHICOPEE", "loc" : [ -72.607962, 42.162046 ],
"pop" : 23396, "state" : "MA" }
{ "_id" : "01020", "city" : "CHICOPEE", "loc" : [ -72.576142, 42.176443 ],
"pop" : 31495, "state" : "MA" }
```

```
jsondf = spark.read.json("/mnt/GoFast/bronze/zips.json")
display(jsondf)
```

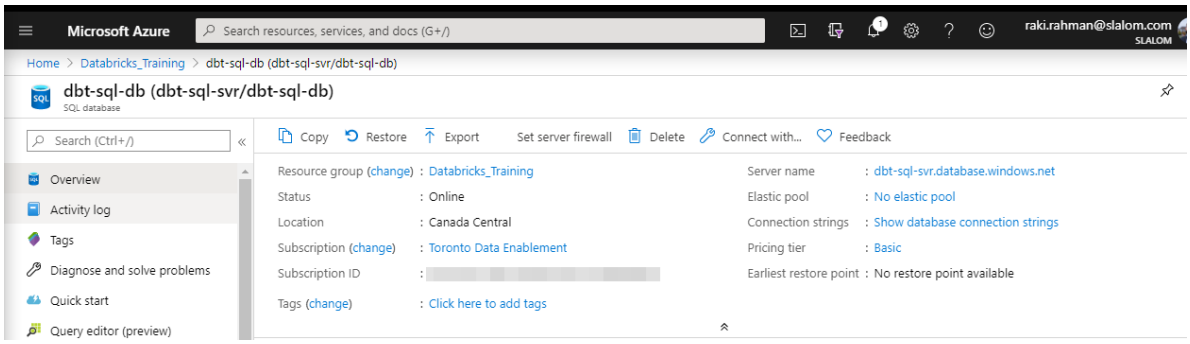
_id	city
01001	AGAWAM
01002	CUSHMAN
01005	BARRE
01007	BELCHERTOWN
01008	BLANDFORD
01010	BRIMFIELD
01011	CHESTER
01012	CHESTERFIELD
01013	CHICOPEE

Showing the first 1000 rows.

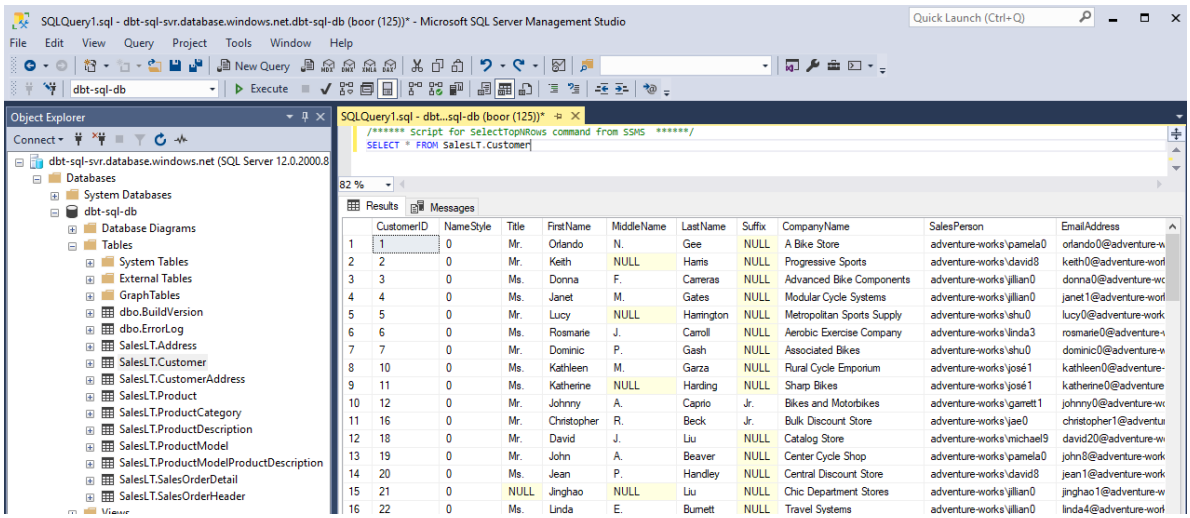
JDBC

We are going to be reading a table from this Azure SQL Database for this activity.

Azure SQL DB



Reading table in SQL Server Management Studio



Set up JDBC connection:

```
jdbcUsername = "your--SQL--username"
jdbcPassword = "your--SQL--password"
driverClass = "com.microsoft.sqlserver.jdbc.SQLServerDriver"
jdbcHostname = "your-sql-svr.database.windows.net"
jdbcPort = 1433
jdbcDatabase = "your-sql-db"

# Create the JDBC URL without passing in the user and password parameters.
jdbcUrl = "jdbc:sqlserver://{0}:{1};database={2}".format(jdbcHostname,
jdbcPort, jdbcDatabase)

# Create a Properties() object to hold the parameters.
connectionProperties = {
  "user" : jdbcUsername,
  "password" : jdbcPassword,
  "driver" : driverClass
}
```

Run Query against JDBC connection:

```
pushdown_query = "(SELECT * FROM SalesLT.Customer) Customers"
jdbcdf = spark.read.jdbc(url=jdbcUrl, table=pushdown_query,
properties=connectionProperties)
display(jdbcdf)
```

CustomerID	NameStyle	Title	FirstName	MiddleName	LastName	Suffix
1	false	Mr.	Orlando	N.	Gee	null
2	false	Mr.	Keith	null	Harris	null
3	false	Ms.	Donna	F.	Carreras	null

ORC

The **Optimized Row Columnar (ORC)** file format provides a highly efficient way to store Hive data. It was designed to overcome limitations of the other Hive file formats. Using ORC files improves performance when Hive is reading, writing, and processing data.

Let's read `TestVectorOrcFile.testLzo.orc` from our mount.

```
orcdf = spark.read.orc('/mnt/GoFast/bronze/TestVectorOrcFile.testLzo.orc')
display(orcdf)
```

x	y
-1182032251	0
-507358776	1
-2028256754	2
-443757842	3
1300375533	4

1714819092	5
-1487245780	6

Showing the first 1000 rows.

Parquet

Apache Parquet is a free and open-source column-oriented data storage format of the Apache Hadoop ecosystem. It is similar to the other columnar-storage file formats available in Hadoop namely RCFile and ORC. It is compatible with most of the data processing frameworks in the Hadoop environment. It provides efficient data compression and encoding schemes with enhanced performance to handle complex data in bulk.

Parquet is a columnar format that is supported by many other data processing systems. Spark SQL provides support for both reading and writing Parquet files that automatically preserves the schema of the original data. When writing Parquet files, all columns are automatically converted to be nullable for compatibility reasons.

Let's read `wine.parquet` from our mount.

```
parquetDF = spark.read.parquet("/mnt/GoFast/bronze/wine.parquet")
display(parquetDF)
```

fixed_acidity	volatile_acidity	citric_acid	residual_sugar	chlorides
6.1	0.125	0.25	3.3	0.04
3.8	0.31	0.02	11.1	0.036
7.1	0.32	0.34	14.5	0.039
7.2	0.32	0.24	5.6	0.033
6.7	0.25	0.23	7.2	0.038
6.8	0.32	0.32	8.7	0.029
6.6	0.36	0.24	0.9	0.038
6.5	0.28	0.26	8.8	0.04
6.9	0.38	0.28	8.3	0.062

Showing the first 1000 rows.

Text

Let's read `tweets.txt` from our mount, then use a similar command as a `csv` earlier to load into DataFrame.

```
%fs head "dbfs:/mnt/GoFast/bronze/tweets.txt"
```

```
Text= RT @jose_garde: Hospitals use #bigdata platform to improve care, bench
mark quality and manage workflow - http://t.co/412uwH0sqX
Text= Hot off the press: Julius Akinyemi: Africa needs to harness big data t
o really innovate http://t.co/rIXXx26WtH #BigData
Text= Searching big data faster http://t.co/BuR7ei3FPD http://t.co/3lvEU80HR
f , johnny heath corpus christi
Text= #ISAO collaboration with #DHS , ICIT and Federal #IT Leaders. #threat
#bigdata #leadership http://t.co/kcQZbauZoZ
Text= Wacom Bamboo Spark puts your notes on paper on your smartphone http://
t.co/rgnclrEdOX
Text= RT @EurekaMag: Spark generating properties of electrode gels used duri
ng defibrillation a potential fire hazard http://t.co/LjuMsVKi4I
Text= RT @Debrahliani: No matter how buff someone is, if there's no spark ju
st don't force it Lool
Text= “Real Time Analytics With Spark Streaming and Cassandra” https://t.co/
ST5ARPVTMi
Text= RT @SkyDoesTweeting: The Boys head back to school to cause more chaos,
BY HEADING TO SPARKLEZ UNIVERSITY! http://t.co/ewdvLqBhfY
Text= RT @don4luv001: u_t=9x^2u_xx+(9x + 5t)u_x+0u with u(x,s)=abs(3*x^5+3)
http://t.co/I5Csa1rlEz #bigdata #analytics http://t.co/7crPQkmu4z"
Text= Demand for Big Data Analytics Software Still Accelerating @infomgmt ht
```

```
# Delimit on '='
txtdf = (spark.read
         .option("header", "false")
         .option("delimiter", "=")
         .csv("/mnt/GoFast/bronze/tweets.txt")
        ).toDF("Text","Tweet")

# Display "Tweet" Column only
display(txtdf.select("Tweet"))
```

Tweet
RT @jose_garde: Hospitals use #bigdata platform to improve care, benchmark quality and manage w
Hot off the press: Julius Akinyemi: Africa needs to harness big data to really innovate http://t.co/rIXXx2
Searching big data faster http://t.co/BuR7ei3FPD http://t.co/3lvEU80HRf , johnny heath corpus christi
#ISAO collaboration with #DHS , ICIT and Federal #IT Leaders. #threat #bigdata #leadership http://t.c
Wacom Bamboo Spark puts your notes on paper on your smartphone http://t.co/rgnclrEdOX

RT @EurekaMag: Spark generating properties of electrode gels used during defibrillation a potential f
RT @Debrahliani: No matter how buff someone is, if there's no spark just don't force it Lool
“Real Time Analytics With Spark Streaming and Cassandra” https://t.co/ST5ARRVTMi

Tables

Let's read from one of our default Databricks Tables from the Hive Metastore:

```
tabledf = spark.sql("""SELECT * FROM databricks.citydata""")
display(tabledf)
```

rankIn2016	state	stateAbbrev
1	New York	NY
2	California	CA
3	Illinois	IL
4	Texas	TX
5	Arizona	AZ
6	Pennsylvania	PA
7	Texas	TX
8	California	CA
9	Texas	TX

6B) How to configure options for specific formats

Let's read `searose-env.csv` from our mount, specify the schema, header and delimiter.

```
# Schema
Schema = StructType([
    StructField("UTC_date_time", StringType(), True),
    StructField("Unix_UTC_timestamp", FloatType(), True),
    StructField("latitude", FloatType(), True),
    StructField("longitude", FloatType(), True),
    StructField("callsign", StringType(), True),
    StructField("wind_from", StringType(), True),
    StructField("knots", StringType(), True),
    StructField("gust", StringType(), True),
    StructField("barometer", StringType(), True),
    StructField("air_temp", StringType(), True),
    StructField("dew_point", StringType(), True),
    StructField("water_temp", StringType(), True)
])

WeatherDF = (spark.read                                    # The
DataFrameReader
             .option("header", "true")                     # Use first line
of file as header
             .schema(Schema)                               # Enforce Schema
             .option("delimiter", ",")                     # Set delimiter
to ,
             .csv("/mnt/GoFast/bronze/searose-env.csv")    # Creates a
DataFrame from CSV after reading in the file
            )

display(WeatherDF)
```

UTC_date_time	Unix_UTC_timestamp	latitude	longitude
2019-Nov-28 1200	1574942460	46.7	-48
2019-Nov-28 0900	1574931580	46.7	-48
2019-Nov-27 1500	1574866820	46.7	-48
2019-Nov-27 1200	1574855940	46.7	-48
2019-Nov-27 0900	1574845180	46.7	-48
2019-Nov-26 1200	1574769660	46.7	-48
2019-Nov-26 0900	1574758780	46.7	-48
2019-Nov-25 1500	1574694020	46.7	-48
2019-Nov-25 1200	1574683140	46.7	-48

Showing the first 1000 rows.

6C) How to read data from non-core formats using `format()` and `load()`

We can read the same file above `searose-env.csv` with `.format()` and `.load()` syntax - as well as other "non-core" files.

```
Weather2DF = (spark.read
              .format("csv")
              .schema(Schema)
              .option("header","true")
              .option("delimiter", ",")
              .load("/mnt/GoFast/bronze/searose-env.csv")
             )
display(Weather2DF)
```

UTC_date_time	Unix_UTC_timestamp	latitude	longitude
2019-Nov-28 1200	1574942460	46.7	-48
2019-Nov-28 0900	1574931580	46.7	-48
2019-Nov-27 1500	1574866820	46.7	-48
2019-Nov-27 1200	1574855940	46.7	-48
2019-Nov-27 0900	1574845180	46.7	-48
2019-Nov-26 1200	1574769660	46.7	-48
2019-Nov-26 0900	1574758780	46.7	-48
2019-Nov-25 1500	1574694020	46.7	-48
2019-Nov-25 1200	1574683140	46.7	-48

Showing the first 1000 rows.

6D) How to specify a `DDL` -formatted schema

DDL - Data Definition Language

Data Definition Language (DDL) is a standard for commands that define the different structures in a database. DDL statements `create`, `modify`, and `remove` database objects such as `tables`, `indexes`, and `users`. Common DDL statements are `CREATE`, `ALTER`, and `DROP`.

We want to be able to specify our Schema in a DDL format prior to reading a file. Let's try this for our `searose-env.csv` above

```
SchemaDDL = """UTC_date_time VARCHAR(255),
               Unix_UTC_timestamp float,
               latitude float,
               longitude float,
               callsign VARCHAR(255),
               wind_from VARCHAR(255),
               knots VARCHAR(255),
               gust VARCHAR(255),
               barometer VARCHAR(255),
               air_temp VARCHAR(255),
               dew_point VARCHAR(255),
               water_temp VARCHAR(255)"""

Weather3DF = spark.read.csv('/mnt/GoFast/bronze/searose-env.csv',
header=True, schema=SchemaDDL)
display(Weather3DF)
```

UTC_date_time	Unix_UTC_timestamp	latitude	longitude
2019-Nov-28 1200	1574942460	46.7	-48
2019-Nov-28 0900	1574931580	46.7	-48
2019-Nov-27 1500	1574866820	46.7	-48
2019-Nov-27 1200	1574855940	46.7	-48
2019-Nov-27 0900	1574845180	46.7	-48
2019-Nov-26 1200	1574769660	46.7	-48
2019-Nov-26 0900	1574758780	46.7	-48
2019-Nov-25 1500	1574694020	46.7	-48
2019-Nov-25 1200	1574683140	46.7	-48

Showing the first 1000 rows.

6E) How to construct and specify a schema using the `StructType` classes

We've done this above already for task `6B` .

7. DataFrameWriter

Candidates are expected to know how to:

7A) Write data to the “core” data formats (csv, json, jdbc, orc, parquet, text and tables)

Let's create a DataFrame `CustomerAddressDF` from the table `SalesLT.CustomerAddress` in our Azure SQL Database. We will then write this to various different formats to our Storage Account's `silver` Container.

```
# Mount "silver" storage container
mountStorageContainer(storageAccountName,storageAccountAccessKey,"silver","/
mnt/GoFast/silver")
```

```
Mounting silver to /mnt/GoFast/silver:
....Container is already mounted.
```

```
pushdown_query = "(SELECT * FROM SalesLT.CustomerAddress) CustomerAddresses"
CustomerAddressDF = spark.read.jdbc(url=jdbcUrl, table=pushdown_query,
properties=connectionProperties)
display(CustomerAddressDF)
```

CustomerID	AddressID	AddressType	rov
29485	1086	Main Office	16
29486	621	Main Office	22
29489	1069	Main Office	A0
29490	887	Main Office	F1

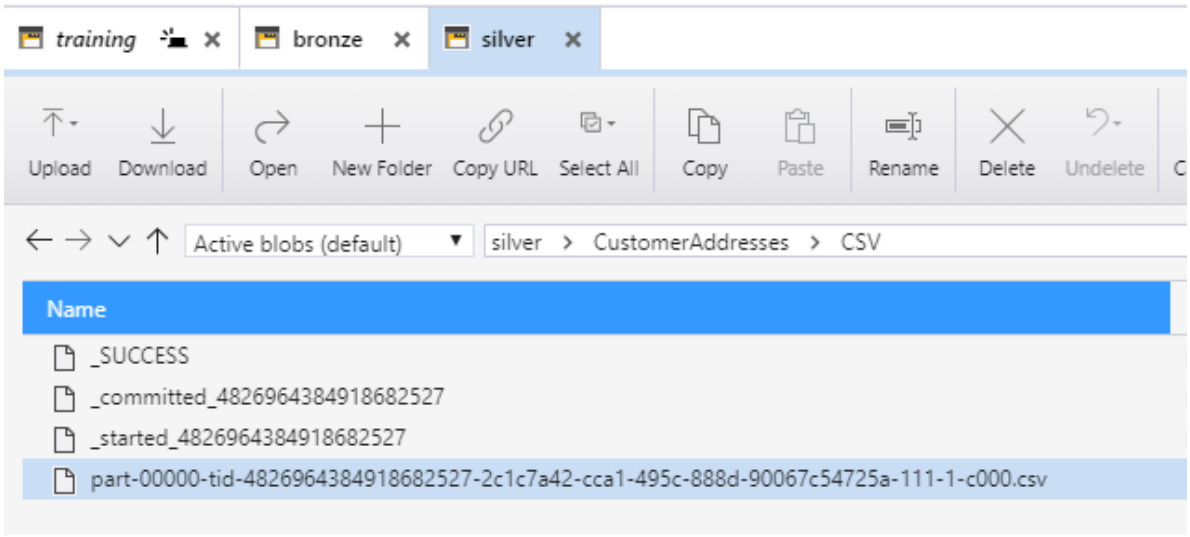
29492	618	Main Office	5B
29494	537	Main Office	49
29496	1072	Main Office	0A

CSV

```
# Clean up directory
dbfsDirPath = "/mnt/GoFast/silver/CustomerAddresses/CSV"
dbutils.fs.rm(dbfsDirPath, recurse=True)
dbutils.fs.mkdirs(dbfsDirPath)

# Write DataFrame to path with header
CustomerAddressDF.write.format("csv").option("header","true").mode("overwrit
e").save(dbfsDirPath)
```

We see this file in our Storage Account:



And the contents are equal to the length of the data received from Azure SQL:

```
30108,550,Main Office,1CAE94D5-5527-4EAC-A1EE-61B828784DD1,2006-08-01T00:00:00.000Z
30109,826,Main Office,CD073E68-C151-4BDE-B286-CF56484CF98C,2006-07-01T00:00:00.000Z
30110,1047,Main Office,63A7A2E0-E0B5-4DA1-8A78-372081EB517E,2006-09-01T00:00:00.000Z
30111,598,Main Office,B7C03322-5A8C-44BC-8B3E-456D16C125E0,2005-08-01T00:00:00.000Z
30112,870,Main Office,18F1221C-EA54-48C2-AD9A-7736FF1010CB,2006-08-01T00:00:00.000Z
30113,653,Main Office,B3D67684-4F0A-4F71-B5E2-2E5E93CCC91C,2006-09-01T00:00:00.000Z
30115,499,Main Office,A02A45B2-673D-4FF6-9C60-EE59A8C2BB4B,2006-08-01T00:00:00.000Z
30116,1044,Main Office,E953D4FE-212C-4A45-9B1F-575CC75969C5,2007-07-01T00:00:00.000Z
30117,596,Main Office,5D215B08-8954-48D2-8254-447D2415FC8B,2005-08-01T00:00:00.000Z
30118,595,Main Office,4ACAF1C5-6E59-43D6-8421-76C9A22FF457,2006-09-01T00:00:00.000Z
```

JSON

```
# Clean up directory
dbfsDirPath = "/mnt/GoFast/silver/CustomerAddresses/JSON"
dbutils.fs.rm(dbfsDirPath, recurse=True)
dbutils.fs.mkdirs(dbfsDirPath)

# Write DataFrame to path
CustomerAddressDF.write.format("json").mode("overwrite").save(dbfsDirPath)
```

JDBC

This time, we do an `overwrite` to our Azure SQL Database Table, by restoring `CustomerAddresses` to a DataFrame from our bucket.

```
# Restore a backup of our DataFrame from Silver Zone
CustomerAddressBackupDF =
spark.read.csv("/mnt/GoFast/silver/CustomerAddresses/CSV", header=True,
inferSchema=True)

# Create temporary view
CustomerAddressBackupDF.createOrReplaceTempView("CustomerAddressView")

# Perform overwrite on SQL table
spark.sql("""SELECT * FROM CustomerAddressView""").write \
    .format("jdbc") \
    .mode("overwrite")  \
    .option("url", jdbcUrl) \
    .option("dbtable", "SalesLT.CustomerAddress") \
    .option("user", jdbcUsername) \
    .option("password", jdbcPassword) \
    .save()
```

ORC

```
# Clean up directory
dbfsDirPath = "/mnt/GoFast/silver/CustomerAddresses/ORC"
dbutils.fs.rm(dbfsDirPath, recurse=True)
dbutils.fs.mkdirs(dbfsDirPath)

# Write DataFrame to path
CustomerAddressDF.write.format("orc").mode("overwrite").save(dbfsDirPath)
```

Parquet

```
# Clean up directory
dbfsDirPath = "/mnt/GoFast/silver/CustomerAddresses/Parquet"
dbutils.fs.rm(dbfsDirPath, recurse=True)
dbutils.fs.mkdirs(dbfsDirPath)

# Write DataFrame to path
CustomerAddressDF.write.format("parquet").mode("overwrite").save(dbfsDirPath
)
```

Text

```
# Clean up directory
dbfsDirPath = "/mnt/GoFast/silver/CustomerAddresses/Text"
dbutils.fs.rm(dbfsDirPath, recurse=True)
dbutils.fs.mkdirs(dbfsDirPath)

# Note that text file does not support `int` data type, and also expects one
column, so we must convert to one '|'' seperated string
CustomerAddressConcatDF = CustomerAddressDF \
                                    .withColumn("CustomerID",
col("CustomerID").cast("string")) \
                                    .withColumn("AddressID",
col("AddressID").cast("string")) \
                                    .withColumn("ModifiedDate",
col("ModifiedDate").cast("string")) \
                                    .withColumn("Concatenated",
concat(col("CustomerID"), lit('|'), \

col("AddressID"), lit('|'), \

col("AddressType"), lit('|'), \

col("rowguid"), lit('|'), \

col("ModifiedDate"))) \
                                    .drop(col("CustomerID")) \
                                    .drop(col("AddressID")) \
                                    .drop(col("AddressType")) \
                                    .drop(col("rowguid")) \
                                    .drop(col("ModifiedDate"))

# Write DataFrame to path
CustomerAddressConcatDF.write.format("text").mode("overwrite").save(dbfsDirP
ath)
```

Tables

Let's create a Hive table usign `Parquet` (location above).

```
%sql

USE databricks;

DROP TABLE IF EXISTS CustomerAddress;
CREATE TABLE IF NOT EXISTS CustomerAddress
USING parquet
OPTIONS  (path "/mnt/GoFast/silver/CustomerAddresses/Parquet");

SELECT COUNT(*) FROM databricks.CustomerAddress
```

count(1)
417

7B) Overwriting existing files

We've already achieved this above when persisting a DataFrame to mount, by using `.mode("overwrite")` - we also do this below.

7C) How to configure options for specific formats

Previously, we wrote a CSV file with the `CustomerAddress` data that was `,` seperated. Let's overwrite that file with a `|` seperated version now.

Output:

```
30108|550|Main Office|1CAE94D5-5527-4EAC-A1EE-61B828784DD1|2006-08-01T00:00:00.000Z
30109|826|Main Office|CD073E68-C151-4BDE-B286-CF56484CF98C|2006-07-01T00:00:00.000Z
30110|1047|Main Office|63A7A2E0-E0B5-4DA1-8A78-372081EB517E|2006-09-01T00:00:00.000Z
30111|598|Main Office|B7C03322-5A8C-44BC-8B3E-456D16C125E0|2005-08-01T00:00:00.000Z
30112|870|Main Office|18F1221C-EA54-48C2-AD9A-7736FF1010CB|2006-08-01T00:00:00.000Z
30113|653|Main Office|B3D67684-4F0A-4F71-B5E2-2E5E93CCC91C|2006-09-01T00:00:00.000Z
30115|499|Main Office|A02A45B2-673D-4FF6-9C60-EE59A8C2BB4B|2006-08-01T00:00:00.000Z
30116|1044|Main Office|E953D4FE-212C-4A45-9B1F-575CC75969C5|2007-07-01T00:00:00.000Z
30117|596|Main Office|5D215B08-8954-48D2-8254-447D2415FC8B|2005-08-01T00:00:00.000Z
30118|595|Main Office|4ACAF1C5-6E59-43D6-8421-76C9A22FF457|2006-09-01T00:00:00.000Z

```

```
# Specify CSV directory
dbfsDirPath = "/mnt/GoFast/silver/CustomerAddresses/CSV"

# Write DataFrame to path with header and seperator
CustomerAddressDF.write.format("csv").option("header","true").option("delimi
ter", "|").mode("overwrite").save(dbfsDirPath)
```

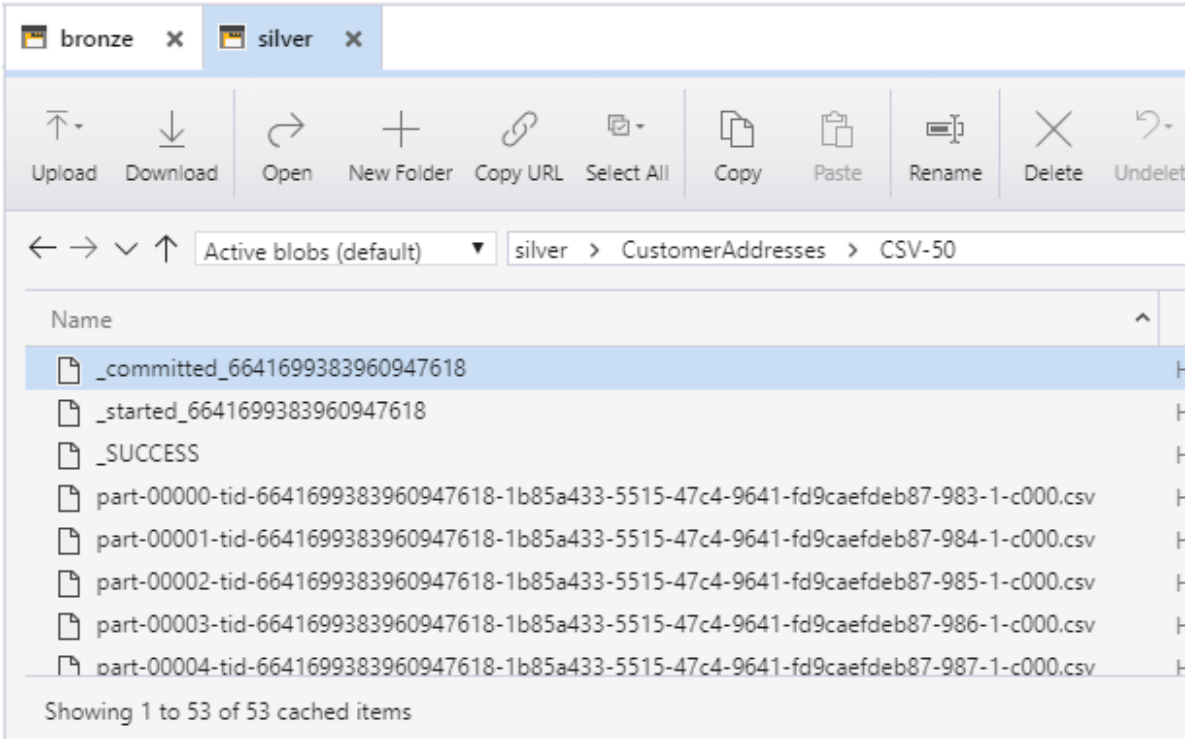
7D) How to write a data source to 1 single file or N separate files

Difference between `coalesce` and `repartition`

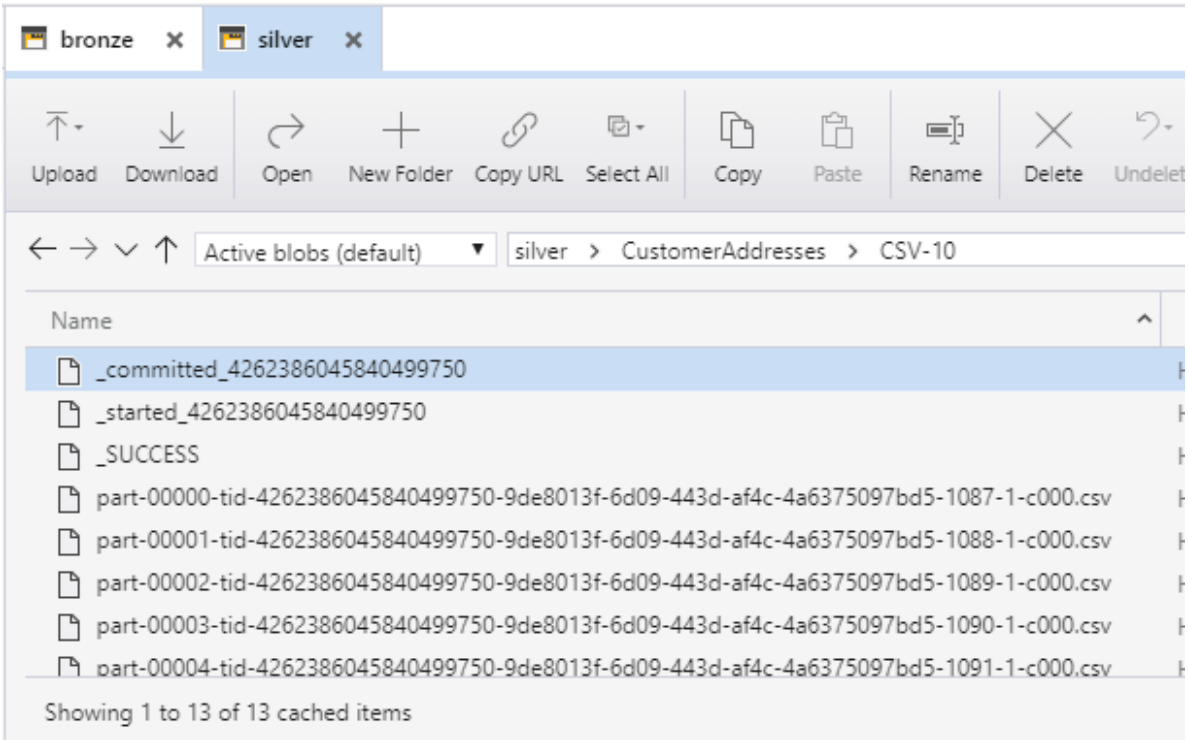
`repartition()` literally reshuffles the data to form as many partitions as we specify, i.e. the number of partitions can be **increased/decreased**. Whereas with `coalesce()` we avoid data movement and use the existing partitions, meaning the number of partitions can only be **decreased**.

Note that `coalesce()` results in partitions with different amounts of data per partition, whereas `repartition()` is distributed evenly. As a result, the `coalesce()` operation may run faster than `repartition()`, but the partitions themselves may work slower further on because Spark is built to work with equal sized partitions across the task slots on the executors.

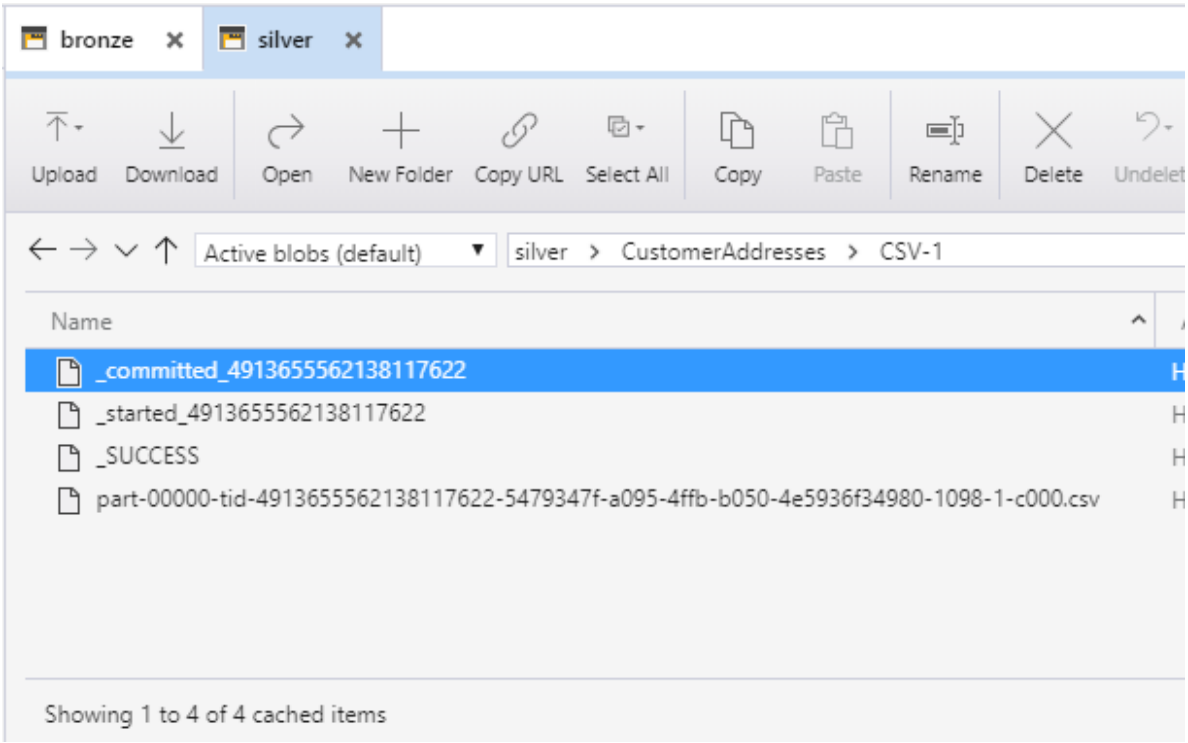
50 partitions:



10 partitions:



1 partition:


bronze
silver
Upload
Download
Open
New Folder
Copy URL
Select All
Copy
Paste
Rename
Delete
Active blobs (default)
silver > CustomerAddresses > CSV-1
Name
_committed_4913655562138117622
_started_4913655562138117622
_SUCCESS
part-00000-tid-4913655562138117622-5479347f-a095-4ffb-b050-4e5936f34980-1098-1-c000.csv
Showing 1 to 4 of 4 cached items

```
# Clean up directory for 50 part CSV
dbfsDirPath = "/mnt/GoFast/silver/CustomerAddresses/CSV-50"
dbutils.fs.rm(dbfsDirPath, recurse=True)
dbutils.fs.mkdirs(dbfsDirPath)

# Redefine DataFrame with 50 partitions
CustomerAddressDF = CustomerAddressDF.repartition(50)

# Write DataFrame to path with header and repartition to 50 partitions
CustomerAddressDF.write.format("csv").option("header","true").mode("overwrit
e").save(dbfsDirPath)

# Clean up directory for 10 part CSV
dbfsDirPath = "/mnt/GoFast/silver/CustomerAddresses/CSV-10"
dbutils.fs.rm(dbfsDirPath, recurse=True)
dbutils.fs.mkdirs(dbfsDirPath)

# Write DataFrame to path with header and coalesce to 10 partitions
CustomerAddressDF.coalesce(10).write.format("csv").option("header","true").m
ode("overwrite").save(dbfsDirPath)

# Clean up directory for 1 part CSV
dbfsDirPath = "/mnt/GoFast/silver/CustomerAddresses/CSV-1"
dbutils.fs.rm(dbfsDirPath, recurse=True)
dbutils.fs.mkdirs(dbfsDirPath)

# Write DataFrame to path with header and coalesce to 1 partition
CustomerAddressDF.coalesce(1).write.format("csv").option("header","true").mo
de("overwrite").save(dbfsDirPath)
```

7E) How to write partitioned data

Partitioning by Columns

Partitioning uses **partitioning columns** to divide a dataset into smaller chunks (based on the values of certain columns) that will be written into separate directories.

With a partitioned dataset, Spark SQL can load only the parts (partitions) that are really needed (and avoid doing filtering out unnecessary data on JVM). That leads to faster load time and more efficient memory consumption which gives a better performance overall.

With a partitioned dataset, Spark SQL can also be executed over different subsets (directories) in parallel at the same time.

```
display(CustomerAddressDF)
```

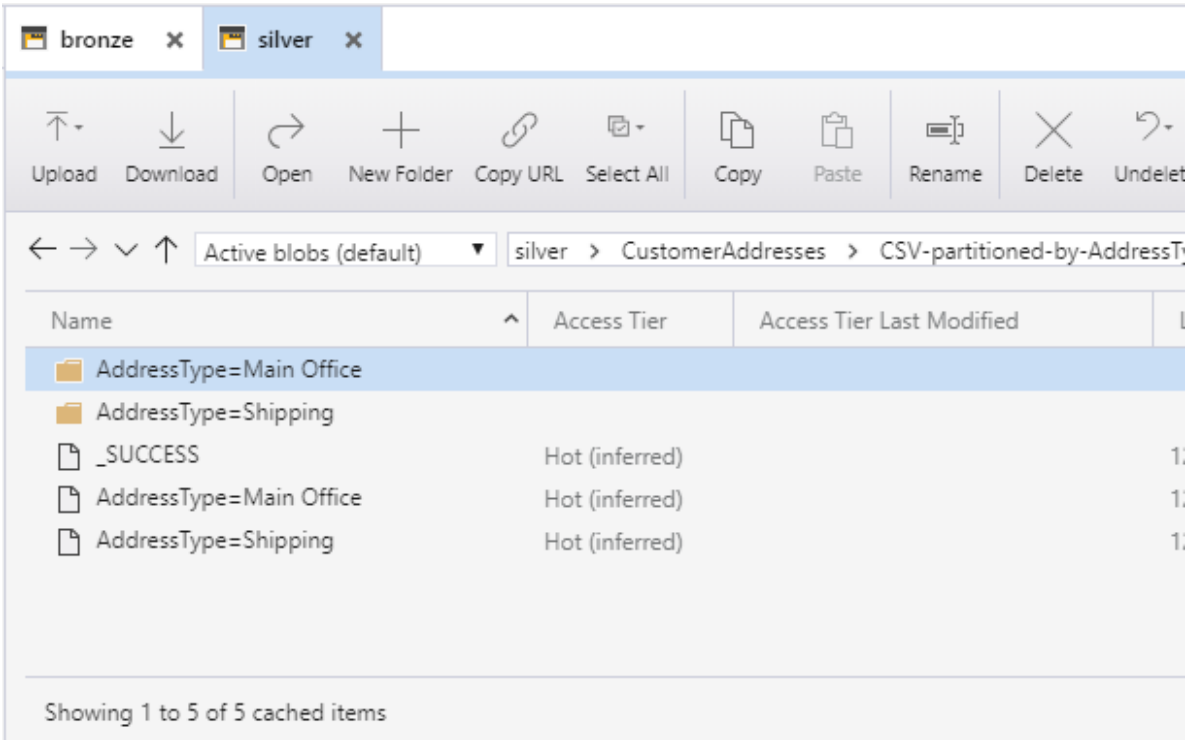
CustomerID	AddressID	AddressType	rov
29766	522	Main Office	3D
29932	637	Main Office	16
29690	1036	Main Office	C2
30051	888	Main Office	A3
30053	507	Main Office	44
29734	517	Main Office	AB
29982	1102	Main Office	95
29794	466	Main Office	B3
29620	503	Main Office	C7

Given our dataset above, let's partition by `AddressType` and write to storage.

```
# Clean up directory
dbfsDirPath = "/mnt/GoFast/silver/CustomerAddresses/CSV-partitioned-by-AddressType"
dbutils.fs.rm(dbfsDirPath, recurse=True)
dbutils.fs.mkdirs(dbfsDirPath)

# Write DataFrame to path with header and partition by AddressType
CustomerAddressDF.write.format("csv").option("header","true").mode("overwrite").partitionBy("AddressType").save(dbfsDirPath)
```

And we see:



7F) How to bucket data by a given set of columns

What is Bucketing?

Bucketing is an optimization technique that uses **buckets** (and **bucketing columns**) to determine data partitioning and avoid data shuffle.

The motivation is to optimize performance of a join query by avoiding shuffles (aka *exchanges*) of tables participating in the join. Bucketing results in fewer exchanges (and so stages).

```
# Note that this is only supported to a table (and not to a location)
CustomerAddressDF.write \
  .mode("overwrite") \
  .bucketBy(10, "ModifiedDate") \
  .saveAsTable("CustomerAddress_bucketed")

display(sql('''SELECT * FROM CustomerAddress_bucketed'''))
```

CustomerID	AddressID	AddressType	ro
29690	1036	Main Office	C2
29734	517	Main Office	AE
29890	855	Main Office	17
30106	630	Main Office	16
30067	464	Main Office	AF
29533	801	Main Office	CI
29883	11381	Shipping	C4
29994	482	Main Office	50
29810	835	Main Office	60

8. DataFrame

8A) Have a working understanding of every action such as `take()`, `collect()`, and `foreach()`

Let's demo the following table of **Actions** on our DataFrame:

Method	Return	Description
`collect()`	Collection	Returns an array that contains all of Rows in this Dataset.
`count()`	Long	Returns the number of rows in the Dataset.
`first()`	Row	Returns the first row.
`foreach(f)`	-	Applies a function f to all rows.
`foreachPartition(f)`	-	Applies a function f to each partition of this Dataset.
`head()`	Row	Returns the first row.
`reduce(f)`	Row	Reduces the elements of this Dataset using the specified binary function.
`show(..)`	-	Displays the top 20 rows of Dataset in a tabular form.

Method	Return	Description
`take(n)`	Collection	Returns the first n rows in the Dataset.
`toLocalIterator()`	Iterator	Return an iterator that contains all of Rows in this Dataset.

Note once again that while **Transformations** always return a *DataFrame*, **Actions** either return a *result* or *write to disk*.

`collect()`

Return a list that contains all of the elements in this RDD.

Note: This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

```
Array = CustomerAddressDF.collect()
print(Array[0])
print("\n")
print(Array[0][0])
```

```
Row(CustomerID=29766, AddressID=522, AddressType='Main Office', rowguid='3DF
9AFB4-CCE1-4E5B-87B4-01AC35D30607', ModifiedDate=datetime.datetime(2007, 2,
1, 0, 0))


29766
```

`count()`

Return the number of elements in this RDD.

```
CustomerAddressDF.count()
```

```
Out[41]: 417
```

`first()`

Return the first element in this RDD/DataFrame.

```
CustomerAddressDF.first()
```

```
Out[42]: Row(CustomerID=29766, AddressID=522, AddressType='Main Office', row
guid='3DF9AFB4-CCE1-4E5B-87B4-01AC35D30607', ModifiedDate=datetime.datetime
```

```
(2007, 2, 1, 0, 0))
```

`foreach(f)`

Applies a function `f` to all elements of this RDD/DataFrame.

Note: `RDD.foreach` method runs on the cluster for each *worker*, and so we don't see the print output.

```
def f(x): print(x)
CustomerAddressDF.foreach(f)
```

`foreachPartition(f)`

Applies a function to each partition of this RDD.

Note: `RDD.foreachPartition` method runs on the cluster for each *worker*, and so we don't see the print output.

```
def f(x): print(x)
CustomerAddressDF.foreachPartition(f)
```

`head()`

Returns the first row.

At first glance, this looks identical to `first()` - let's look at the difference:

Sorted Data

If your data is ***sorted*** using either `sort()` or `ORDER BY` , these operations will be deterministic and return either the 1st element using first()/head() or the top-n using head(n)/take(n).

show()/show(n) return Unit (void) and will print up to the first 20 rows in a tabular form.

These operations may require a shuffle if there are any aggregations, joins, or sorts in the underlying query.

Unsorted Data

If the data is ***not sorted***, these operations are not guaranteed to return the 1st or top-n elements - and a shuffle may not be required.

`show()` / `show(n)` return Unit (void) and will print up to 20 rows in a tabular form and in no particular order.

If no shuffle is required (no aggregations, joins, or sorts), these operations will be optimized to inspect enough partitions to satisfy the operation - likely a much smaller subset of the overall partitions of the dataset.

```
CustomerAddressDF.head()
```

```
Out[45]: Row(CustomerID=29766, AddressID=522, AddressType='Main Office', row
guid='3DF9AFB4-CCE1-4E5B-87B4-01AC35D30607', ModifiedDate=datetime.datetime
(2007, 2, 1, 0, 0))
```

`reduce(f)`

Reduces the elements of this **RDD** (note that it doesn't work on DataFrames) using the specified commutative and associative binary operator. Currently reduces partitions locally.

```
from operator import add
sc.parallelize([2, 4, 6]).reduce(add)
```

```
Out[46]: 12
```

`show(id)`

Displays the top 20 (*default, can be overwritten*) rows of Dataset in a tabular form

```
CustomerAddressDF.show()
```

```
+----------+---------+-----------+--------------------+-------------------+
|CustomerID|AddressID|AddressType|             rowguid|       ModifiedDate|
+----------+---------+-----------+--------------------+-------------------+
|     29766|      522|Main Office|3DF9AFB4-CCE1-4E5...|2007-02-01 00:00:00|
|     29932|      637|Main Office|16E0667F-AA85-415...|2007-09-01 00:00:00|
|     29690|     1036|Main Office|C2213F36-134B-487...|2005-09-01 00:00:00|
|     30051|      888|Main Office|A303B277-ECC4-49D...|2007-07-01 00:00:00|
|     30053|      507|Main Office|4409D7B8-5701-410...|2007-09-01 00:00:00|
|     29734|      517|Main Office|ABDAAE4A-7F6D-419...|2005-07-01 00:00:00|
|     29982|     1102|Main Office|95FD4C03-F298-40A...|2005-12-01 00:00:00|
|     29794|      466|Main Office|B3B568EC-526E-442...|2005-08-01 00:00:00|
|     29620|      503|Main Office|C7A8A997-0817-42D...|2005-08-01 00:00:00|
```

```
|     29890|      855|Main Office|176EF850-4BF4-45C...|2005-07-01 00:00:00|
|     29638|      989|Main Office|3DB4D2FF-611F-416...|2007-09-01 00:00:00|
|     30070|      845|Main Office|4D612D21-D01C-494...|2007-07-01 00:00:00|
|     29560|     1030|Main Office|4012A207-360D-418...|2006-08-01 00:00:00|
|     29853|     1056|Main Office|FE9C02C8-7FF9-4A9...|2005-08-01 00:00:00|
|     30106|      630|Main Office|164491B5-2A57-4FF...|2005-09-01 00:00:00|
|     29524|      599|Main Office|75724647-82A0-49C...|2007-08-01 00:00:00|
|     29765|      806|Main Office|3C5778FE-2189-4E9...|2006-08-01 00:00:00|
|     29492|      618|Main Office|5B3B3EB2-3F43-47E...|2006-12-01 00:00:00|
|     29573|     1016|Main Office|BE9052DE-38C3-405...|2006-09-01 00:00:00|
|     29911|      538|Main Office|B7CDC62A-5672-432...|2007-08-01 00:00:00|
+----------+---------+-----------+--------------------+-------------------+
only showing top 20 rows
```

`take(n)`

Take the first **n** elements of the RDD.

It works by first scanning one partition, and using the results from that partition to estimate the number of additional partitions needed to satisfy the limit.

Note: this method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.

```
CustomerAddressDF.take(2)
```

```
Out[48]: [Row(CustomerID=29766, AddressID=522, AddressType='Main Office', ro
wguid='3DF9AFB4-CCE1-4E5B-87B4-01AC35D30607', ModifiedDate=datetime.datetime
(2007, 2, 1, 0, 0)),
 Row(CustomerID=29932, AddressID=637, AddressType='Main Office', rowguid='16
E0667F-AA85-4155-AB36-1703FC9E07F0', ModifiedDate=datetime.datetime(2007, 9,
1, 0, 0))]
```

`toLocalIterator()`

Return an iterator that contains all of the elements in this RDD/DataFrame. The iterator will consume as much memory as the largest partition in this RDD.

```
[x for x in CustomerAddressDF.toLocalIterator()]
```

```
Out[49]: [Row(CustomerID=29766, AddressID=522, AddressType='Main Office', ro
wguid='3DF9AFB4-CCE1-4E5B-87B4-01AC35D30607', ModifiedDate=datetime.datetime
(2007, 2, 1, 0, 0)),
 Row(CustomerID=29932, AddressID=637, AddressType='Main Office', rowguid='16
E0667F-AA85-4155-AB36-1703FC9E07F0', ModifiedDate=datetime.datetime(2007, 9,
```

```
1, 0, 0)),
 Row(CustomerID=29690, AddressID=1036, AddressType='Main Office', rowguid='C
2213F36-134B-4872-AEC0-2F5EC24F7BCA', ModifiedDate=datetime.datetime(2005,
9, 1, 0, 0)),
 Row(CustomerID=30051, AddressID=888, AddressType='Main Office', rowguid='A3
03B277-ECC4-49D1-AA81-C39D5193D035', ModifiedDate=datetime.datetime(2007, 7,
1, 0, 0)),
 Row(CustomerID=30053, AddressID=507, AddressType='Main Office', rowguid='44
09D7B8-5701-4103-BC8E-60043723F8AA', ModifiedDate=datetime.datetime(2007, 9,
1, 0, 0)),
 Row(CustomerID=29734, AddressID=517, AddressType='Main Office', rowguid='AB
DAAE4A-7F6D-4192-A162-495646CDA3CB', ModifiedDate=datetime.datetime(2005, 7,
1, 0, 0)),
 Row(CustomerID=29982, AddressID=1102, AddressType='Main Office', rowguid='9
5FD4C03-F298-40A1-A4D2-2B02E221D876', ModifiedDate=datetime.datetime(2005, 1
```

8B) Have a working understanding of the various transformations and how they work such as producing a `distinct` set, `filter` ing data, re `partition` ing and `coalesce` ing, performing `join` s and `union` s as well as producing `aggregates`

`select(*cols)`

Projects a set of expressions and returns a new `DataFrame` .

Parameters:

- **cols** – list of column names (`string`) or expressions (`Column`). If one of the column names is `*` , that column is expanded to include all columns in the current DataFrame.

Using our `CustomerAddressDF` from above, let's `select` out a couple of the rows only.

```
TruncatedDF = CustomerAddressDF.select("CustomerID", "rowguid")
display(TruncatedDF)
```

CustomerID	rowguid
29772	BF40660E-40B6-495B-99D0-753(
29877	961322F7-4900-4419-9267-C58A
29897	32AC1E35-A59F-4776-BA25-E64
29699	FF640E5B-57F6-496B-98DA-EB2
30029	4093BD23-27FF-4F33-A21F-6B43
29805	3596CEF3-C2AE-4CE6-95DB-5F
29990	245A36B0-5ABB-41C1-83B6-E99
29696	AEB69976-7375-4C24-B433-952F
29987	88D1880D-F045-40CC-A9CF-51(

`distinct()`

Returns a new `DataFrame` containing the distinct rows in this `DataFrame`.

Let's create a DF with duplicates, and then run `distinct` on it.

```
DuplicateDF = TruncatedDF.withColumn("CustomerID",
lit(29772)).withColumn("rowguid", lit("BF40660E-40B6-495B-99D0-
753CE987B1D1"))
display(DuplicateDF)
```

CustomerID	rowguid
29772	BF40660E-40B6-495B-99D0-75
29772	BF40660E-40B6-495B-99D0-75
29772	BF40660E-40B6-495B-99D0-75
29772	BF40660E-40B6-495B-99D0-75
29772	BF40660E-40B6-495B-99D0-75
29772	BF40660E-40B6-495B-99D0-75
29772	BF40660E-40B6-495B-99D0-75
29772	BF40660E-40B6-495B-99D0-75
29772	BF40660E-40B6-495B-99D0-75

```
display(DuplicateDF.distinct())
```

CustomerID	rowguid
29772	BF40660E-40B6-495B-99D0-75

`groupBy(*cols)`

Groups the `DataFrame` using the specified columns, so we can run aggregation on them. See `GroupedData` for all the available aggregate functions.

`groupby()` is an alias for `groupBy()`.

Parameters:

- **cols** – list of columns to group by. Each element should be a column name (`string`) or an expression (`Column`).

Note: `groupBy` by itself doesn't return anything quantitative, we also have to `agg` by an aggregation function to get a tangible result back.

```
CustomerAddressAggDF =
CustomerAddressDF.groupBy("AddressType").agg({'AddressType':'count'})
display(CustomerAddressAggDF)
```

AddressType
Main Office
Shipping

`sum(*cols)`

Compute the sum for each numeric columns for each group.

Parameters:

- **cols** – list of column names (`string`). Non-numeric columns are ignored.

```
# Let's rename the column "count(AddressType)"
CustomerAddressAggDF2 = CustomerAddressAggDF.withColumn("AddressTypeCount",
col("count(AddressType)")).drop(col("count(AddressType)"))

# We do a groupBy followed by a sum - to ultimately get back our number of
rows in the original DataFrame
display(CustomerAddressAggDF2.groupBy().sum("AddressTypeCount"))
```

sum(AddressTypeCount)
417

`orderBy(*cols, **kwargs)`

Returns a new `DataFrame` sorted by the specified column(s).

Parameters:

- **cols** – list of `Column` or column names to sort by.
- **ascending** – boolean or list of boolean (default True). Sort ascending vs. descending. Specify list for multiple sort orders. If a list is specified, length of the list must equal length of the cols.

```
display(CustomerAddressDF.orderBy("ModifiedDate", ascending = 0))
```

CustomerID	AddressID	AddressType	ro
29831	552	Main Office	A7
29870	484	Main Office	5C
29978	671	Main Office	B3
29948	572	Main Office	7E
29738	610	Main Office	63
30080	885	Main Office	3D
29653	1019	Main Office	1D
29553	581	Main Office	63
29932	627	Main Office	1C

`filter()`

Filters rows using the given condition.

where() is an alias for **filter()**.

Parameters:

- **condition** – a Column of types.BooleanType or a string of SQL expression.

```
display(CustomerAddressDF.filter("AddressID = 484"))
```

CustomerID	AddressID	AddressType	r
29870	484	Main Office	5

`limit()`

Limits the result count to the number specified.

```
display(CustomerAddressDF.limit(3))
```

CustomerID	AddressID	AddressType	r
29766	522	Main Office	3
29932	637	Main Office	1
29690	1036	Main Office	C

`partition() and coalesce`

We already discussed this in detail earlier in **7D**.

`join(other, on=None, how=None)`

Joins with another `DataFrame`, using the given join expression.

Parameters:

- **other** – Right side of the join
- **on** – a string for the join column name, a list of column names, a join expression (Column), or a list of Columns. If on is a string or a list of strings

indicating the name of the join column(s), the column(s) must exist on both sides, and this performs an equi-join.

- **how** – str, default `inner`. Must be one of: `inner`, `cross`, `outer`, `full`, `full_outer`, `left`, `left_outer`, `right`, `right_outer`, `left_semi`, and `left_anti`.

```
# Let's query Azure SQL directly DataFrame we want to recreate
pushdown_query = """(SELECT DISTINCT C.CompanyName, A.City
                    FROM [SalesLT].[Customer] C
                    INNER JOIN [SalesLT].[CustomerAddress] CA ON
C.CustomerID = CA.CustomerID
                    INNER JOIN [SalesLT].[Address] A ON CA.AddressID =
A.AddressID
                    WHERE CA.AddressID IS NOT NULL AND A.AddressLine1 IS NOT
NULL) CompanyAndAddress"""

CompanyAndAddressDF = spark.read.jdbc(url=jdbcUrl, table=pushdown_query,
properties=connectionProperties)
display(CompanyAndAddressDF.orderBy("CompanyName", ascending = 1))
```

CompanyName
A Bike Store
A Great Bicycle Company
A Typical Bike Shop
Acceptable Sales & Service
Action Bicycle Specialists
Active Life Toys
Active Systems
Advanced Bike Components
Aerobic Exercise Company

```
# Get the underlying Tables as DataFrames from Azure SQL
pushdown_query = "(SELECT CustomerID, CompanyName FROM SalesLT.Customer)
Customer"
CustomerDF = spark.read.jdbc(url=jdbcUrl, table=pushdown_query,
properties=connectionProperties)

pushdown_query = "(SELECT CustomerID, AddressID FROM
SalesLT.CustomerAddress) CustomerAddress"
CustomerAddressDF = spark.read.jdbc(url=jdbcUrl, table=pushdown_query,
properties=connectionProperties)

pushdown_query = "(SELECT AddressID,City FROM SalesLT.Address) Address"
AddressDF = spark.read.jdbc(url=jdbcUrl, table=pushdown_query,
properties=connectionProperties)


# Perform joins identical to goal DataFrame above
display(CustomerDF.join(CustomerAddressDF, CustomerDF.CustomerID ==
CustomerAddressDF.CustomerID, 'inner') \
                  .join(AddressDF, CustomerAddressDF.AddressID ==
AddressDF.AddressID, 'inner') \
                  .select(CustomerDF.CompanyName, AddressDF.City) \
                  .orderBy("CompanyName", ascending = 1) \
                  .distinct())
```

CompanyName
A Bike Store
A Great Bicycle Company
A Typical Bike Shop
Acceptable Sales & Service
Action Bicycle Specialists
Active Life Toys
Active Systems
Advanced Bike Components
Aerobic Exercise Company

And we get back the same DataFrame as `CompanyAndAddressDF` from our above SQL query.

union()

Return a new `DataFrame` containing union of rows in this and another frame.

This is equivalent to *UNION ALL* in SQL. To do a SQL-style set union (that does deduplication of elements), use this function followed by `distinct()` .

Also as standard in SQL, this function resolves columns by position (not by name).

```
# Create two DataFrames with fruit lists, with some multiple occurences
df1 = spark.createDataFrame(["apple", "orange", "apple", "mango"],
StringType())
df2 = spark.createDataFrame(["cherries", "orange", "blueberry", "apple"],
StringType())

# Perform union and display data
df3 = df1.union(df2)
display(df3)
```

value
apple
orange
apple
mango
cherries
orange
blueberry
apple

agg(*exprs)

Compute aggregates and returns the result as a `DataFrame` .

The available aggregate functions can be:

1. built-in aggregation functions, such as *avg, max, min, sum, count*
2. group aggregate pandas UDFs, created with
   `pyspark.sql.functions.pandas_udf()`

Note: There is no partial aggregation with group aggregate UDFs, i.e., a full shuffle is required. Also, all the data of a group will be loaded into memory, so the user should be aware of the potential OOM risk if data is skewed and certain groups are too large to fit in memory.

```
# Perform aggregation on `count` of the fruit names - i.e. the number of
occurences in the union list per fruit.
display(df3.groupby("value").agg({'value':'count'}).orderBy("value",
ascending = 1))
```

value	cou
apple	3
blueberry	1
cherries	1
mango	1
orange	2

8C) Know how to `cache` data, specifically to `disk`, `memory` or both

Caching in Spark

As we discussed earlier, Spark supports pulling data sets into a cluster-wide in-memory `cache`. This is very useful when data is accessed repeatedly, such as when querying a small “hot” dataset (required in many operations) or when running an iterative algorithm.

Here are some relevant functions:

`cache()`

Persists the `DataFrame` with the default storage level (`MEMORY_AND_DISK`). `persist()` without an argument is equivalent with `cache()`.

`unpersist()`

Marks the `DataFrame` as non-persistent, and remove all blocks for it from memory and disk.

`persist(storageLevel)`

Sets the storage level to persist the contents of the `DataFrame` across operations after the first time it is computed. This can only be used to assign a new storage level if the `DataFrame` does not have a storage level set yet. If no storage level is specified defaults to (`MEMORY_AND_DISK`).

As we see below, there are 4 caching levels that can be fine-tuned with `persist()` . Let's look into the options available in more detail.

Storage Level	Equivalent	Descrip
`MEMORY_ONLY`	`StorageLevel(False, True, False, False, 1)`	Store I deseri objects the RD memo partitic cache recom each ti neede defaul
`MEMORY_AND_DISK`	`StorageLevel(True, True, False, False, 1)`	Store I deseri objects the RD memo partitic on disl from th neede
`DISK_ONLY`	`StorageLevel(True, False, False, False, 1)`	Store t partitic
`MEMORY_ONLY_2`	`StorageLevel(False, True, False, False, 2)`	Same above each p cluster
`MEMORY_AND_DISK_2`	`StorageLevel(True, True, False, False, 2)`	Same above each p cluster
`OFF_HEAP`	`StorageLevel(True, True, True, False, 1)`	Simila MEMC but sto heap r require memo

As a simple example, let's mark our `CompanyAndAddressDF` dataset to be cached. While it may seem silly to use Spark to explore and cache a small `DataFrame`, the interesting part is that these same functions can be used on very large data sets, even when they are striped across tens or hundreds of nodes.

```
# Check the cache level before we do anything
CompanyAndAddressDF.storageLevel

Out[63]: StorageLevel(False, False, False, False, 1)

# Let's cache the DataFrame with default - MEMORY_AND_DISK, and check the
cache level
CompanyAndAddressDF.cache().storageLevel

Out[64]: StorageLevel(True, True, False, True, 1)

# Unpersist and proceed to Memory Only Cache
CompanyAndAddressDF.unpersist()
CompanyAndAddressDF.persist(storageLevel =
StorageLevel.MEMORY_ONLY).storageLevel

Out[65]: StorageLevel(False, True, False, False, 1)

# Unpersist and proceed to Memory and Disk 2 Cache
CompanyAndAddressDF.unpersist()
CompanyAndAddressDF.persist(storageLevel =
StorageLevel.MEMORY_AND_DISK_2).storageLevel

Out[66]: StorageLevel(True, True, False, False, 2)
```

And so on.

8D) Know how to uncache previously cached data

```
# Let's uncache the DataFrame
CompanyAndAddressDF.unpersist()

# Check the Cache level
CompanyAndAddressDF.storageLevel

Out[67]: StorageLevel(False, False, False, False, 1)
```

8E) Converting a *DataFrame* to a global or temp view.

Global Temporary View vs. Temporary View

Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates. If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view. Global temporary view is tied to a system preserved database `global_temp` , and we must use the qualified name to refer it, e.g. `SELECT * FROM global_temp.view1` . We use these two commands:

- **Global Temp View**: `df.createOrReplaceGlobalTempView("tempViewName")` creates a global temporary view with this dataframe df. Lifetime of this view is dependent to spark application itself, if you want to drop this view: `spark.catalog.dropGlobalTempView("tempViewName")`
- **Temp View**: `df.createOrReplaceTempView("tempViewName")` creates or replaces a local temporary view with this dataframe df. Lifetime of this view is dependent to SparkSession class, if you want to drop this view: `spark.catalog.dropTempView("tempViewName")`

```
# Create Global Temporary View
CompanyAndAddressDF.createOrReplaceGlobalTempView("CompanyAndAddressGlobal")
display(spark.sql("SELECT * FROM global_temp.CompanyAndAddressGlobal"))
```

CompanyName
A Bike Store
A Great Bicycle Company
A Typical Bike Shop
Acceptable Sales & Service
Action Bicycle Specialists
Active Life Toys
Active Systems
Advanced Bike Components
Aerobic Exercise Company

```
# Create SQL Temporary View
CompanyAndAddressDF.createOrReplaceTempView("CompanyAndAddressTemp")
display(spark.sql("SELECT * FROM CompanyAndAddressTemp"))
```

CompanyName
A Bike Store
A Great Bicycle Company
A Typical Bike Shop
Acceptable Sales & Service
Action Bicycle Specialists
Active Life Toys
Active Systems
Advanced Bike Components
Aerobic Exercise Company

```
# Drop Temporary Views
spark.catalog.dropGlobalTempView("CompanyAndAddressGlobal")
spark.catalog.dropTempView("CompanyAndAddressTemp")
```

8F) Applying hints

Structured queries can be optimized using **Hint Framework** that allows for specifying query hints.

Query hints allow for annotating a query and give a hint to the query optimizer how to optimize logical plans. This can be very useful when the query optimizer cannot make optimal decision, e.g. with respect to join methods due to conservativeness or the lack of proper statistics.

Spark SQL supports COALESCE and REPARTITION and BROADCAST hints. All remaining unresolved hints are silently removed from a query plan at analysis.

```
display(CustomerDF.join(CustomerAddressDF.hint("broadcast"), "CustomerID"))
```

CustomerID	CompanyName
29485	Professional Sales and Service

29486	Riders Company
29489	Area Bike Accessories
29490	Bicycle Accessories and Kits
29492	Valley Bicycle Specialists
29494	Vinyl and Plastic Goods Corporation
29496	Fun Toys and Bikes

9. Row & Column

9A) Candidates are expected to know how to work with row and columns to successfully extract data from a *DataFrame*

We can easily get back a single Column or Row and perform manipulations in Spark. For example, with our DataFrame `CustomerDF`：

```
display(CustomerDF)
```

CustomerID	CompanyName
1	A Bike Store
2	Progressive Sports
3	Advanced Bike Compon
4	Modular Cycle Systems
5	Metropolitan Sports Sup
6	Aerobic Exercise Compa
7	Associated Bikes
10	Rural Cycle Emporium
11	Sharp Bikes

```
# Extract First Row
display(CustomerDF.filter("CompanyName == 'A Bike Store' and CustomerID == 1"))
```

CustomerID
1

```
# Extract Distinct Second Column 'CompanyName', Not like "A **** ****", Order By descending, Pick Top 5
display(CustomerDF.select("CompanyName").filter("CompanyName not like 'A%'").distinct().orderBy("CompanyName", ascending = 1).take(5))
```

CompanyName
Basic Bike Company
Basic Sports Equipment
Beneficial Exercises and Activities
Best o' Bikes
Bicycle Accessories and Kits

10. Spark SQL Functions

When instructed what to do, candidates are expected to be able to employ the multitude of Spark SQL functions. Examples include, but are not limited to:

Let's use this Databricks table `databricks.citydata` for this section wherever applicable:

```
%sql

SELECT * FROM databricks.citydata
```

rankIn2016	state	stateAbbrev
1	New York	NY
2	California	CA
3	Illinois	IL

4	Texas	TX
5	Arizona	AZ
6	Pennsylvania	PA
7	Texas	TX

10A) Aggregate functions: getting the `first` or `last` item from an array or computing the `min` and `max` values of a column.

`first(expr[, isIgnoreNull])`

Returns the first value of expr for a group of rows. If `isIgnoreNull` is true, returns only non-null values.

```
%sql

SELECT first(state), last(state) FROM
  (SELECT state FROM  databricks.citydata
  ORDER BY state ASC)
```

first(state, false)
Alaska

`min(col)`

Aggregate function: returns the minimum value of the expression in a group.

```
%sql

SELECT min(estPopulation2016), max(estPopulation2016) FROM
  (SELECT estPopulation2016 FROM  databricks.citydata)
```

min(estPopulation2016)
100392

10B) Collection functions: testing if an array `contains` a value, `explode`ing or `flatten`ing data.

```
%sql
SELECT array_contains(array('a','b','c','d'), 'a') AS contains_a;
```

contains_a
true

```
%sql
SELECT explode(array('a','b','c','d'));
```

col
a
b
c
d

```
%sql
SELECT flatten(array(array(1, 2), array(3, 4)));
```

flatten(array(array(1, 2), array(3, 4)))
▸ [1,2,3,4]

10C) Date time functions: parsing `string`s into `timestamp`s or formatting `timestamp`s into `string`s

`to_date(date_str[, fmt])`

Parses the `date_str` expression with the `fmt` expression to a date. Returns `null` with invalid input. By default, it follows casting rules to a date if the `fmt` is omitted.

```
%sql
SELECT to_date('2009-07-30 04:17:52') AS date, to_date('2016-12-31', 'yyyy-MM-dd') AS date_formatted
```

date	da
2009-07-30	20

`to_timestamp(timestamp[, fmt])`

Parses the `timestamp` expression with the `fmt` expression to a `timestamp`. Returns `null` with invalid input. By default, it follows casting rules to a `timestamp` if the fmt is omitted.

```
%sql
SELECT to_timestamp('2016-12-31 00:12:00') AS timestamp, to_timestamp('2016-12-31', 'yyyy-MM-dd') AS timestamp_formatted;
```

timestamp
2016-12-31T00:12:00.000+0000

`date_format(timestamp, fmt)`

Converts `timestamp` to a value of `string` in the format specified by the date format `fmt`.

```
%sql

SELECT date_format('2016-12-31T00:12:00.000+0000', 'y') AS year_only
```

year_only
2016

10D) Math functions: computing the cosign, floor or log of a number

`cos(expr)`

Returns the cosine of `expr`.

```
%sql
SELECT cos(0) AS cos
```

cos
1

`floor(expr)`

Returns the largest integer not greater than `expr`.

```
%sql
SELECT floor(5.9) AS floored
```

floored
5

`log(base, expr)`

Returns the logarithm of `expr` with base.

```
%sql

SELECT log(2, 8) AS log_2
```

log_2
3

10E) Misc functions: converting a value to crc32, md5, sha1 or sha2

`crc32(expr)`

Returns a cyclic redundancy check value of the `expr` as a `bigint` .

```
%sql
SELECT crc32('hello')
```

crc32(CAST(hello AS BINARY))
907060870

`md5(expr)`

Returns an MD5 128-bit checksum as a hex string of `expr` .

```
%sql
SELECT md5('hello')
```

md5(CAST(hello AS BINARY))
5d41402abc4b2a76b9719d911017c592

`sha1(expr)`

Returns a sha1 hash value as a hex string of the `expr`.

```
%sql
SELECT sha1('hello')
```

sha1(CAST(hello AS BINARY))
aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d

`sha2(expr, bitLength)`

Returns a checksum of SHA-2 family as a hex string of expr. SHA-224, SHA-256, SHA-384, and SHA-512 are supported. Bit length of 0 is equivalent to 256.

```
%sql
SELECT sha2('hello', 256)
```

sha2(CAST(hello AS BINARY), 256)
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

10F) Non-aggregate functions: creating an array, testing if a column is null, not-null, nan, etc

`array(expr, ...)`

Returns an array with the given elements.

```
%sql
SELECT array(1, 2, 3) AS array
```

array
▸ [1,2,3]

`isnull(expr)`

Returns `true` if expr is `null` , or `false` otherwise.

```
%sql

SELECT isnull(population2010) FROM databricks.citydata
limit 5
```

(population2010 IS NULL)
false
false
false
false
false

`isnotnull(expr)`

Returns `true` if `expr` is *not* `null` , or `false` otherwise.

```
%sql

SELECT isnotnull(population2010) FROM databricks.citydata
limit 5
```

(population2010 IS NOT NULL)
true
true
true
true
true

Difference between `null` and `NaN`

`null` values represents "no value" or "nothing", it's not even an empty string or zero. It can be used to represent that nothing useful exists.

`NaN` stands for "Not a Number", it's usually the result of a mathematical operation that doesn't make sense, e.g. 0.0/0.0.

`isnan(expr)`
Returns `true` if expr is `NaN` , or `false` otherwise.

```
%sql

SELECT cast('NaN' as double), isnan(cast('NaN' as double)), cast('hello' as double), isnan(cast('hello' as double))
```

CAST(NaN AS DOUBLE)	isnan(CAST(NaN AS DOUBLE))
NaN	true

10G) Sorting functions: sorting data in descending order, ascending order, and sorting with proper null handling

Let's use our `WeatherDF` for this section:

```
display(WeatherDF)
```

UTC_date_time	Unix_UTC_timestamp	latitude	longitude
2019-Nov-28 1200	1574942460	46.7	-48
2019-Nov-28 0900	1574931580	46.7	-48
2019-Nov-27 1500	1574866820	46.7	-48
2019-Nov-27 1200	1574855940	46.7	-48
2019-Nov-27 0900	1574845180	46.7	-48
2019-Nov-26 1200	1574769660	46.7	-48
2019-Nov-26 0900	1574758780	46.7	-48
2019-Nov-25 1500	1574694020	46.7	-48
2019-Nov-25 1200	1574683140	46.7	-48

Showing the first 1000 rows.

```
# We replace the `NULL` in column `gust` and `air_temp` above with `null`,
and typecast to float
WeatherDFtemp = WeatherDF.withColumn("gust",when((ltrim(col("gust")) ==
"NULL"),lit(None)).otherwise(col("gust").cast("float"))) \
                     .withColumn("air_temp",when((ltrim(col("air_temp")) ==
"NULL"),lit(None)).otherwise(col("air_temp").cast("float")))


# Create temp SQL view
WeatherDFtemp.createOrReplaceTempView('Weather')
```

We demonstrate:

- Filter for values where **gust** is not `null` and **air_temp** is not `null`
- Order by **gust** `ascending`
- Order by **air_temp** `descending`

```
%sql

SELECT gust, air_temp
FROM Weather
WHERE isnotnull(gust) AND isnotnull(air_temp)
ORDER BY gust ASC, air_temp DESC
```

gust
3.000002
5.00001
5.00001
6.000004
6.000004
7.0000176
7.0000176
8.000011
9.000026

10H) String functions: applying a provided regular expression, trimming string and extracting substrings.

`regexp_extract(str, regexp[, idx])`

Extracts a group that matches `regexp`.

```
%sql
SELECT regexp_extract('Verified by Stacy', '( by)')
```

regexp_extract(Verified by Stacy, (by), 1)
by

`regexp_replace(str, regexp, rep)`

Replaces all substrings of `str` that match regexp with `rep`.

```
%sql
SELECT regexp_replace('Verified by Stacy', '( by)', ':')
```

regexp_replace(Verified by Stacy, (by), :)
Verified: Stacy

`trim(str)`

Removes the leading and trailing space characters from str.

`trim(BOTH trimStr FROM str)`

Remove the leading and trailing trimStr characters from str

`trim(LEADING trimStr FROM str)`

Remove the leading trimStr characters from str

`trim(TRAILING trimStr FROM str)`

Remove the trailing trimStr characters from str

Arguments:

`str` - a string expression

`trimStr` - the trim string characters to trim, the default value is a single space

`BOTH , FROM` - these are keywords to specify trimming string characters from both ends of the string

`LEADING , FROM` - these are keywords to specify trimming string characters from the left end of the string

`TRAILING , FROM` - these are keywords to specify trimming string characters from the right end of the string

```
%sql
SELECT trim('    SparkSQL   '),
       trim('SL', 'SSparkSQLS'),
       trim(BOTH 'SL' FROM 'SSparkSQLS'),
       trim(LEADING 'SL' FROM 'SSparkSQLS'),
       trim(TRAILING 'SL' FROM 'SSparkSQLS')
```

trim(SparkSQL)	trim(SSparkSQLS, SL)	trim(S
SparkSQL	parkSQ	parkSQ

10l) UDF functions: employing a UDF function.

We call the `squaredWithPython` UDF we defined earlier.

```
%sql
SELECT gust, squaredWithPython(floor(gust))
FROM Weather
WHERE isnotnull(gust)
ORDER BY gust ASC
```

gust	squaredWithPython(FLOOR(cast(gust as double)))
3.000002	9
5.00001	25
5.00001	25
6.000004	36
6.000004	36

7.0000176	49
7.0000176	49

10J) Window functions: computing the rank or dense rank.

Let's create a `DataFrame` to demo these functions, and create a `Window`.

What are `Window` Functions?

A window function calculates a return value for every input row of a table based on a group of rows, called the **Frame**. Every input row can have a unique frame associated with it. This characteristic of window functions makes them more powerful than other functions and allows users to express various data processing tasks that are hard (if not impossible) to be expressed without window functions in a concise way.

```
schema = StructType([
  StructField("letter", StringType(), True),
  StructField("position", IntegerType(), True)])

df = spark.createDataFrame([("a", 10), ("a", 10), ("a", 20)], schema)
windowSpec = Window.partitionBy("letter").orderBy("position")
display(df)
```

letter
a
a
a

`rank()`

Computes the rank of a value in a group of values. The result is one plus the number of rows preceding or equal to the current row in the ordering of the

partition. The values will produce gaps in the sequence.

`dense_rank()`

Computes the rank of a value in a group of values. The result is one plus the previously assigned rank value. Unlike the function `rank`, `dense_rank` will not produce gaps in the ranking sequence.

`row_number()`

Returns a sequential number starting at 1 within a window partition.

```
display(df.withColumn("rank", rank().over(windowSpec))
  .withColumn("dense_rank", dense_rank().over(windowSpec))
  .withColumn("row_number", row_number().over(windowSpec)))
```

letter	position	rank
a	10	1
a	10	1
a	20	3

Note that the value "10" exists twice in **position** within the same window (**letter** = "a"). That's when you see a difference between the three functions `rank`, `dense_rank`, `row`.

Another example

We create a *productRevenue* table as seen below.

We want to answer two questions:

1. What are the best-selling and the second best-selling products in every category?
2. What is the difference between the revenue of each product and the revenue of the best-selling product in the same category of that product?

```
# Create the Products
Product = Row("product", "category", "revenue")


# Create DataFrames from rows
ProductDF = spark.createDataFrame([
                                   Product('Thin', 'Cell phone', 6000),
                                   Product('Normal', 'Tablet', 1500),
                                   Product('Mini', 'Tablet', 5500),
                                   Product('Ultra thin', 'Cell phone',
5500),
                                   Product('Very thin', 'Cell phone', 6000),
                                   Product('Big', 'Tablet', 2500),
                                   Product('Bendable', 'Cell phone', 3000),
                                   Product('Foldable', 'Cell phone', 3000),
                                   Product('Pro', 'Tablet', 4500),
                                   Product('Pro2', 'Tablet', 6500)
                                  ])
ProductDF.createOrReplaceTempView("productRevenue")
display(ProductDF)
```

product	category
Thin	Cell phone
Normal	Tablet
Mini	Tablet
Ultra thin	Cell phone
Very thin	Cell phone
Big	Tablet
Bendable	Cell phone
Foldable	Cell phone
Pro	Tablet

To answer the first question “**What are the best-selling and the second best-selling products in every category?**”, we need to rank *products* in a *category* based on their *revenue*, and to pick the best selling and the second best-selling products based the ranking.

Below is the SQL query used to answer this question by using window function `dense_rank`.

```
%sql

SELECT
  product,
  category,
  revenue
FROM (
  SELECT
    product,
    category,
    revenue,
    dense_rank() OVER (PARTITION BY category ORDER BY revenue DESC) as rank
  FROM productRevenue) tmp
WHERE
  rank <= 2
ORDER BY category DESC
```

product	category
Pro2	Tablet
Mini	Tablet
Thin	Cell phone
Very thin	Cell phone
Ultra thin	Cell phone

For the second question "**What is the difference between the revenue of each product and the revenue of the best selling product in the same category as that product?**", to calculate the revenue difference for a *product*, we need to find the highest revenue value from *products* in the same *category* for each product.

Below is a Python DataFrame program used to answer this question.

product	category
Pro2	Tablet
Mini	Tablet
Pro	Tablet
Big	Tablet
Normal	Tablet
Very thin	Cell phone
Thin	Cell phone