Databricks Spark

```
--Dataframe
budget_df = (spark
.table("products")
.select("name", "price")
.where("price < 200")
.orderBy("price")
)
```

```
display(budget_df) --to output the results of a dataframe.
```

```
The schema defines the column names and types of a dataframe.

Access a dataframe's schema using the schema attribute.

1
budget_df.schema
View a nicer output for this schema using the printSchema() method.

1
budget_df.printSchema()
```

```
--lazily evaluated
(products_df
  .select("name", "price")
  .where("price < 200")
  .orderBy("price"))

--Use actions to trigger computation
(products_df
  .select("name", "price")
  .where("price < 200")
  .orderBy("price")
  .show())
```

DataFrame Action Methods

Aa Method	Description
show	Displays the top n rows of DataFrame in a tabular form
count	Returns the number of rows in the DataFrame
describe, summary	Computes basic statistics for numeric and string columns
first, head	Returns the the first row
collect	Returns an array that contains all rows in this DataFrame
take	Returns an array of the first n rows in the DataFrame
Untitled	

```
-- count returns the number of records in a DataFrame.
budget_df.count()
-- collect returns an array of all rows in a DataFrame.
budget_df.collect()
```

Spark SQL executes all queries on the same engine. Spark SQL optimizes queries before execution. Query plan > Optimized Query Plan > RDDs > Executions

The SparkSession is the single entry point to all DataFrame API functionality. Automatically created in a Databricks notebook as the variable spark

`SparkSession` Methods

Aa Method	Description
sql	Returns a DataFrame representing the result of the given query
table	Returns the specified table as a DataFrame
read	Returns a DataFrameReader that can be used to read data in as a DataFrame
range	Create a DataFrame with a column containing elements in a range from start to end (exclusive) with step value and number of partitions
createDataFrame	Creates a DataFrame from a list of tuples, primarily used for testing

```
Convert between DataFrames and SQL
-- createOrReplaceTempView creates a temporary view based on the DataFrame. The lifetime of the temporary view is tied to the SparkSession
budget_df.createOrReplaceTempView("budget")
display(spark.sql("SELECT * FROM budget"))

Read Parquet - spark.read.parquet("path/to/files")
spark.read.parquet("/mnt/training/ecommerce/events.parquet")

Write Parquet
(df.write
  .option("compression", "snappy")
  .mode("overwrite")
  .parquet(outPath)
)
```

```
-- Read from CSV files

users_csv_path = f"{datasets_dir}/users/users-500k.csv"

users_df = (spark
           .read
           .option("sep", "\t")
           .option("header", True)
           .option("inferSchema", True)
           .csv(users_csv_path)
          )

users_df.printSchema()
-- Spark's Python API also allows you to specify the DataFrameReader options as parameters to the csv method

users_df = (spark
           .read
           .csv(users_csv_path, sep="\t", header=True, inferSchema=True)
          )

users_df.printSchema()

--Manually define the schema by creating a StructType with column names and data types

from pyspark.sql.types import LongType, StringType, StructType, StructField

user_defined_schema = StructType([
    StructField("user_id", StringType(), True),
    StructField("user_first_touch_timestamp", LongType(), True),
    StructField("email", StringType(), True)
])

-- Read from CSV using this user-defined schema instead of inferring the schema

users_df = (spark
           .read
           .option("sep", "\t")
           .option("header", True)
           .schema(user_defined_schema)
           .csv(users_csv_path)
          )
```

```
--Alternatively, define the schema using data definition language (DDL) syntax.
ddl_schema = "user_id string, user_first_touch_timestamp long, email string"

users_df = (spark
           .read
           .option("sep", "\t")
           .option("header", True)
           .schema(ddl_schema)
           .csv(users_csv_path)
          )
```

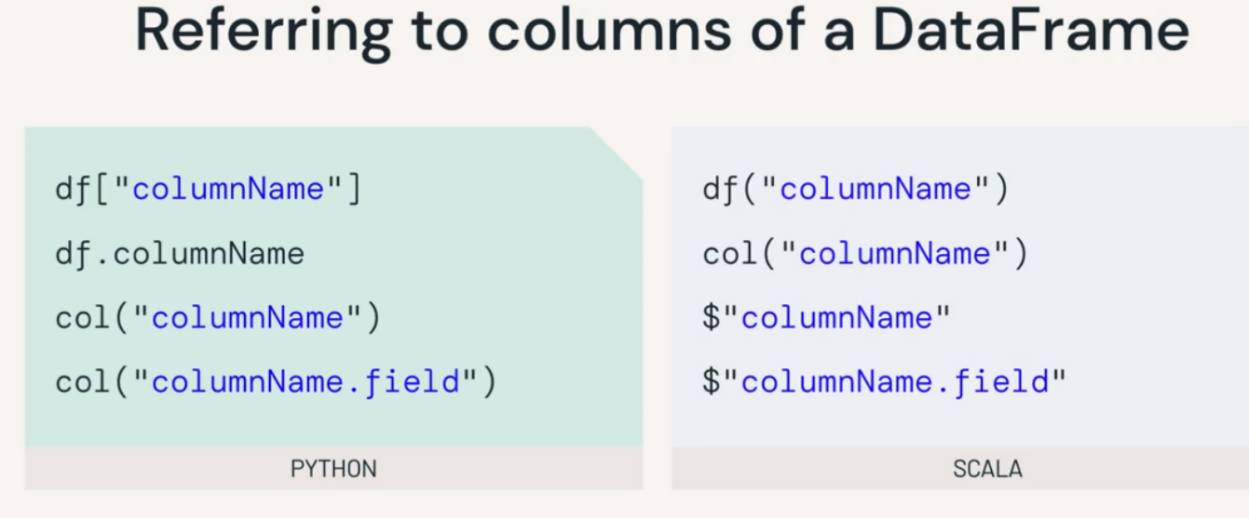
```
-- Read from JSON files

events_json_path = f"{datasets_dir}/events/events-500k.json"

events_df = (spark
            .read
            .option("inferSchema", True)
            .json(events_json_path)
           )

events_df.printSchema()

--Read data faster by creating a StructType with the schema names and data types

from pyspark.sql.types import ArrayType, DoubleType, IntegerType, LongType, StringType, StructType, StructField

user_defined_schema = StructType([
    StructField("device", StringType(), True),
    StructField("ecommerce", StructType([
        StructField("purchaseRevenue", DoubleType(), True),
        StructField("total_item_quantity", LongType(), True),
        StructField("unique_items", LongType(), True)
    ]), True),
    StructField("event_name", StringType(), True),
    StructField("event_previous_timestamp", LongType(), True),
    StructField("event_timestamp", LongType(), True),
    StructField("geo", StructType([
        StructField("city", StringType(), True),
        StructField("state", StringType(), True)
    ]), True),
    StructField("items", ArrayType(
        StructType([
            StructField("coupon", StringType(), True),
            StructField("item_id", StringType(), True),
            StructField("item_name", StringType(), True),
            StructField("item_revenue_in_usd", DoubleType(), True),
            StructField("price_in_usd", DoubleType(), True),
            StructField("quantity", LongType(), True)
        ])
    ), True),
    StructField("traffic_source", StringType(), True),
    StructField("user_first_touch_timestamp", LongType(), True),
    StructField("user_id", StringType(), True)
])

events_df = (spark
            .read
            .schema(user_defined_schema)
            .json(events_json_path)
           )
You can use the StructType Scala method toDDL to have a DDL-formatted string created for you.

This is convenient when you need to get the DDL-formated string for ingesting CSV and JSON but you don't want to hand craft it or the Struc

However, this functionality is not available in Python but the power of the notebooks allows us to use both languages.

# Step 1 - use this trick to transfer a value (the dataset path) between Python and Scala using the shared spark-config
spark.conf.set("com.whatever.your_scope.events_path", events_json_path)

In a Python notebook like this one, create a Scala cell to injest the data and produce the DDL formatted schema
%scala
// Step 2 - pull the value from the config (or copy & paste it)
val eventsJsonPath = spark.conf.get("com.whatever.your_scope.events_path")

// Step 3 - Read in the JSON, but let it infer the schema
val eventsSchema = spark.read
```

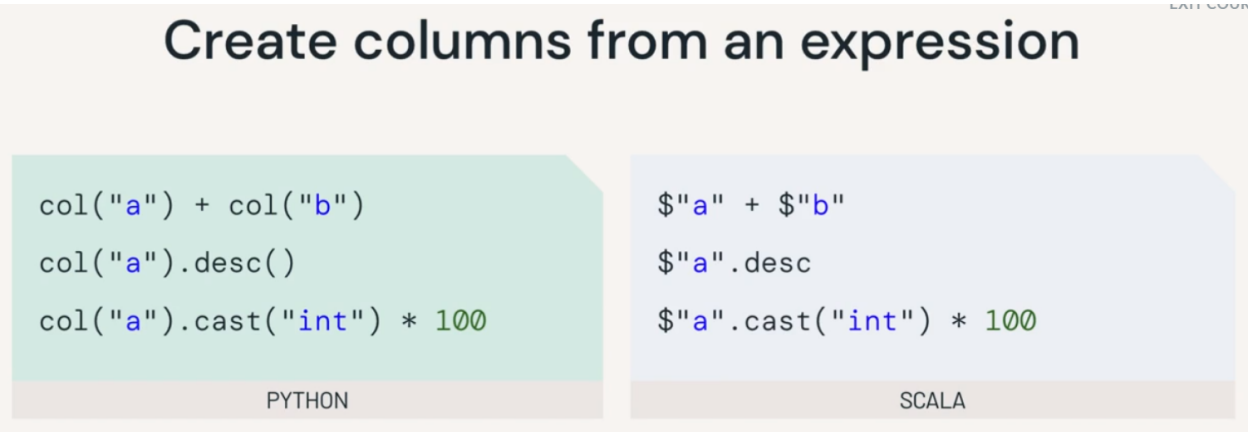
```
                        .option("inferSchema", true)
                        .json(eventsJsonPath)
                        .schema.toDDL

// Step 4 - print the schema, select it, and copy it.
println("="*80)
println(eventsSchema)
println("="*80)

# Step 5 - paste the schema from above and assign it to a variable as seen here
events_schema = "`device` STRING,`ecommerce` STRUCT<`purchase_revenue_in_usd`: DOUBLE, `total_item_quantity`: BIGINT, `unique_items`: BIGIN

# Step 6 - Read in the JSON data using our new DDL formatted string
events_df = (spark.read
                .schema(events_schema)
                .json(events_json_path))

display(events_df)
```

```
DataFrameWriter
Interface used to write a DataFrame to external storage systems

(df
  .write
  .option("compression", "snappy")
  .mode("overwrite")
  .parquet(output_dir)
)

DataFrameWriter is accessible through the SparkSession attribute write. This class includes methods to write DataFrames to different extern

users_output_dir = working_dir + "/users.parquet"

(users_df
 .write
 .option("compression", "snappy")
 .mode("overwrite")
 .parquet(users_output_dir)
)

display(
    dbutils.fs.ls(users_output_dir)
) -> this will show you if command is successful or not

-- As with DataFrameReader, Spark's Python API also allows you to specify the DataFrameWriter options as parameters to the parquet method

(users_df
 .write
 .parquet(users_output_dir, compression="snappy", mode="overwrite")
)
```

```
-- Write DataFrames to tables
-- Write events_df to a table using the DataFrameWriter method saveAsTable

-- Note This creates a global table, unlike the local view created by the DataFrame method createOrReplaceTempView

events_df.write.mode("overwrite").saveAsTable("events")
```

```
-- Write Results to a Delta Table
-- Write events_df with the DataFrameWriter's save method and the following configurations: Delta format & overwrite mode

events_output_path = working_dir + "/delta/events"

(events_df
 .write
 .format("delta")
 .mode("overwrite")
 .save(events_output_path)
)
```





Column Operators and Methods

Method	Description
*, +, <, >=	Math and comparison operators
==, !=	Equality and inequality tests (Scala operators are `===` and `=!=`)
alias	Gives the column an alias
cast, astype	Casts the column to a different data type
isNull, isNotNull, isNan	Is null, is not null, is NaN
asc, desc	Returns a sort expression based on ascending/descending order of the column

```
-- Create complex expressions with existing columns, operators, and methods.

rev_df = (events_df
         .filter(col("ecommerce.purchase_revenue_in_usd").isNotNull())
         .withColumn("purchase_revenue", (col("ecommerce.purchase_revenue_in_usd") * 100).cast("int"))
         .withColumn("avg_purchase_revenue", col("ecommerce.purchase_revenue_in_usd") / col("ecommerce.total_item_quantity"))
         .sort(col("avg_purchase_revenue").desc())
        )
```

DataFrame Transformation Methods

Method	Description
`select`	Returns a new DataFrame by computing given expression for each element
`drop`	Returns a new DataFrame with a column dropped
`withColumnRenamed`	Returns a new DataFrame with a column renamed
`withColumn`	Returns a new DataFrame by adding a column or replacing the existing column that has the same name
`filter`, `where`	Filters rows using the given condition
`sort`, `orderBy`	Returns a new DataFrame sorted by the given expressions
`dropDuplicates`, `distinct`	Returns a new DataFrame with duplicate rows removed
`limit`	Returns a new DataFrame by taking the first n rows
`groupBy`	Groups the DataFrame using the specified columns, so we can run aggregation on them

```
-- Selects a list of columns or column based expressions

devices_df = events_df.select("user_id", "device")
display(devices_df)

from pyspark.sql.functions import col

locations_df = events_df.select(
    "user_id",
    col("geo.city").alias("city"),
    col("geo.state").alias("state")
)
display(locations_df)
```

```
-- selectExpr()
-- Selects a list of SQL expressions

apple_df = events_df.selectExpr("user_id", "device in ('macOS', 'iOS') as apple_user")
display(apple_df)

-- drop()
-- Returns a new DataFrame after dropping the given column, specified as a string or Column object

-- Use strings to specify multiple columns

anonymous_df = events_df.drop("user_id", "geo", "device")
display(anonymous_df)

no_sales_df = events_df.drop(col("ecommerce"))
display(no_sales_df)

-- withColumn()
-- Returns a new DataFrame by adding a column or replacing an existing column that has the same name.

mobile_df = events_df.withColumn("mobile", col("device").isin("iOS", "Android"))
display(mobile_df)

purchase_quantity_df = events_df.withColumn("purchase_quantity", col("ecommerce.total_item_quantity").cast("int"))
purchase_quantity_df.printSchema()

-- withColumnRenamed()
-- Returns a new DataFrame with a column renamed.

location_df = events_df.withColumnRenamed("geo", "location")
display(location_df)

-- filter()
-- Filters rows using the given SQL expression or column based condition.

-- Alias: where

purchases_df = events_df.filter("ecommerce.total_item_quantity > 0")
display(purchases_df)

revenue_df = events_df.filter(col("ecommerce.purchase_revenue_in_usd").isNotNull())
display(revenue_df)

android_df = events_df.filter((col("traffic_source") != "direct") & (col("device") == "Android"))
```

```
-- dropDuplicates()
-- Returns a new DataFrame with duplicate rows removed, optionally considering only a subset of columns.

-- Alias: distinct

display(events_df.distinct())

distinct_users_df = events_df.dropDuplicates(["user_id"])
display(distinct_users_df)

-- limit()
-- Returns a new DataFrame by taking the first n rows.

limit_df = events_df.limit(100)
display(limit_df)

-- sort()
-- Returns a new DataFrame sorted by the given columns or expressions.

-- Alias: orderBy

increase_timestamps_df = events_df.sort("event_timestamp")
display(increase_timestamps_df)

decrease_timestamp_df = events_df.sort(col("event_timestamp").desc())

increase_sessions_df = events_df.orderBy(["user_first_touch_timestamp", "event_timestamp"])
display(increase_sessions_df)

decrease_sessions_df = events_df.sort(col("user_first_touch_timestamp").desc(), col("event_timestamp"))
display(decrease_sessions_df)
```

```
-- Solution Exercisesfi
mac_df = (events_df
          .where("device == 'macOS'")
          .sort("event_timestamp")
         )
THIS IS SAME AS:

mac_sql_df = spark.sql("""
SELECT *
FROM events
WHERE device = 'macOS'
ORDER By event_timestamp
""")

display(mac_sql_df)
```

Grouped data methods

Various aggregation methods are available on the GroupedData object.

Aa Method	Description
agg	Compute aggregates by specifying a series of aggregate columns
avg	Compute the mean value for each numeric columns for each group
count	Count the number of rows for each group
max	Compute the max value for each numeric columns for each group
mean	Compute the average value for each numeric columns for each group
min	Compute the min value for each numeric column for each group
pivot	Pivots a column of the current DataFrame and performs the specified aggregation
sum	Compute the sum for each numeric columns for each group

```
event_counts_df = df.groupBy("event_name").count()
display(event_counts_df)
```

```
avg_state_purchases_df = df.groupBy("geo.state").avg("ecommerce.purchase_revenue_in_usd")
display(avg_state_purchases_df)

city_purchase_quantities_df = df.groupBy("geo.state", "geo.city").sum("ecommerce.total_item_quantity", "ecommerce.purchase_revenue_in_usd")
display(city_purchase_quantities_df)
```

Built-In Functions

In addition to DataFrame and Column transformation methods, there are a ton of helpful functions in Spark's built-in SQL functions module.

In Scala, this is `org.apache.spark.sql.functions`, and `pyspark.sql.functions` in Python. Functions from this module must be imported into your code.

Aggregate Functions

Here are some of the built-in functions available for aggregation.

Aa Method	Description
approx_count_distinct	Returns the approximate number of distinct items in a group
avg	Returns the average of the values in a group
collect_list	Returns a list of objects with duplicates
corr	Returns the Pearson Correlation Coefficient for two columns
max	Compute the max value for each numeric columns for each group
mean	Compute the average value for each numeric columns for each group
stddev_samp	Returns the sample standard deviation of the expression in a group
sumDistinct	Returns the sum of distinct values in the expression
var_pop	Returns the population variance of the values in a group

Use the grouped data method `agg` to apply built-in aggregate functions

This allows you to apply other transformations on the resulting columns, such as `alias`.

```
from pyspark.sql.functions import sum

state_purchases_df = df.groupBy("geo.state").agg(sum("ecommerce.total_item_quantity").alias("total_purchases"))
display(state_purchases_df)

from pyspark.sql.functions import avg, approx_count_distinct

state_aggregates_df = (df
                       .groupBy("geo.state")
                       .agg(avg("ecommerce.total_item_quantity").alias("avg_quantity"),
                            approx_count_distinct("user_id").alias("distinct_users"))
                      )

display(state_aggregates_df)
```

Math Functions

Here are some of the built-in functions for math operations.

Aa Method	Description
ceil	Computes the ceiling of the given column.
cos	Computes the cosine of the given value.
log	Computes the natural logarithm of the given value.
round	Returns the value of the column e rounded to 0 decimal places with HALF_UP round mode.

Method	Description
sqrt	Computes the square root of the specified float value.

```
from pyspark.sql.functions import cos, sqrt

display(spark.range(10)  # Create a DataFrame with a single column called "id" with a range of integer values
        .withColumn("sqrt", sqrt("id"))
        .withColumn("cos", cos("id"))
       )
```

Spark uses pattern letters for date and timestamp parsing and formatting. A subset of these patterns are shown below.

Symbol	Meaning	Presentation	Examples
Untitled			
G	era	text	AD; Anno Domini
y	year	year	2020; 20
D	day-of-year	number(3)	189
M/L	month-of-year	month	7; 07; Jul; July
d	day-of-month	number(3)	28
Q/q	quarter-of-year	number/text	3; 03; Q3; 3rd quarter
E	day-of-week	text	Tue; Tuesday

Built-In Functions: Date Time Functions

Here are a few built-in functions to manipulate dates and times in Spark.

Method	Description
`add_months`	Returns the date that is numMonths after startDate
`current_timestamp`	Returns the current timestamp at the start of query evaluation as a timestamp column
`date_format`	Converts a date/timestamp/string to a value of string in the format specified by the date format given by the second argument.
`dayofweek`	Extracts the day of the month as an integer from a given date/timestamp/string
`from_unixtime`	Converts the number of seconds from unix epoch (1970-01-01 00:00:00 UTC) to a string representing the timestamp of that moment in the current system time zone in the yyyy-MM-dd HH:mm:ss format
`minute`	Extracts the minutes as an integer from a given date/timestamp/string.
`unix_timestamp`	Converts time string with given pattern to Unix timestamp (in seconds)

```
-- cast()
-- Casts column to a different data type, specified using string representation or DataType.

timestamp_df = df.withColumn("timestamp", (col("timestamp") / 1e6).cast("timestamp"))
display(timestamp_df)

from pyspark.sql.types import TimestampType

timestamp_df = df.withColumn("timestamp", (col("timestamp") / 1e6).cast(TimestampType()))
display(timestamp_df)

-- date_format()
-- Converts a date/timestamp/string to a string formatted with the given date time pattern.

from pyspark.sql.functions import date_format

formatted_df = (timestamp_df
                .withColumn("date string", date_format("timestamp", "MMMM dd, yyyy"))
                .withColumn("time string", date_format("timestamp", "HH:mm:ss.SSSSSS"))
               )
display(formatted_df)
```

```
-- Extract datetime attribute from timestamp
-- year
-- Extracts the year as an integer from a given date/timestamp/string.
-- Similar methods: month, dayofweek, minute, second, etc.

from pyspark.sql.functions import year, month, dayofweek, minute, second

datetime_df = (timestamp_df
               .withColumn("year", year(col("timestamp")))
               .withColumn("month", month(col("timestamp")))
               .withColumn("dayofweek", dayofweek(col("timestamp")))
               .withColumn("minute", minute(col("timestamp")))
               .withColumn("second", second(col("timestamp")))
              )
display(datetime_df)

-- Convert to date
-- to_date
-- Converts the column into DateType by casting rules to DateType.

from pyspark.sql.functions import to_date

date_df = timestamp_df.withColumn("date", to_date(col("timestamp")))
display(date_df)

-- Manipulate Datetimes
-- date_add
-- Returns the date that is the given number of days after start

from pyspark.sql.functions import date_add

plus_2_df = timestamp_df.withColumn("plus_two_days", date_add(col("timestamp"), 2))
display(plus_2_df)
```

```
Transformations -> Complex Type (Hard to understand, might need rewatch)

from pyspark.sql.functions import *
df = spark.read.format("delta").load(sales_path)
display(df)
# You will need this DataFrame for a later exercise
details_df = (df
              .withColumn("items", explode("items"))
              .select("email", "items.item_name")
              .withColumn("details", split(col("item_name"), " "))
             )
display(details_df)
```

String Functions

Here are some of the built-in functions available for manipulating strings.

Aa Method	Description
translate	Translate any character in the src by a character in replaceString
regexp_replace	Replace all substrings of the specified string value that match regexp with rep
regexp_extract	Extract a specific group matched by a Java regex, from the specified string column
ltrim	Removes the leading space characters from the specified string column
lower	Converts a string column to lowercase
split	Splits str around matches of the given pattern

For example: let's imagine that we need to parse our `email` column. We're going to use the `split` function to split domain and handle.

```
from pyspark.sql.functions import split
display(df.select(split(df.email, '@', 0).alias('email_handle')))
```

Collection Functions

Here are some of the built-in functions available for working with arrays.

Method	Description
array_contains	Returns null if the array is null, true if the array contains value, and false otherwise.
element_at	Returns element of array at given index. Array elements are numbered starting with **1**.
explode	Creates a new row for each element in the given array or map column.
collect_set	Returns a set of objects with duplicate elements eliminated.

```
mattress_df = (details_df
               .filter(array_contains(col("details"), "Mattress"))
               .withColumn("size", element_at(col("details"), 2)))
display(mattress_df)
```

Aggregate Functions

Here are some of the built-in aggregate functions available for creating arrays, typically from GroupedData.

Method	Description
collect_list	Returns an array consisting of all values within the group.
collect_set	Returns an array consisting of all unique values within the group.

```
-- Let's say that we wanted to see the sizes of mattresses ordered by each email address. For this, we can use the collect_set function
size_df = mattress_df.groupBy("email").agg(collect_set("size").alias("size options"))

display(size_df)
```

Union and unionByName

The DataFrame `union` method resolves columns by position, as in standard SQL. You should use it only if the two DataFrames have exactly the same schema, including the column order. In contrast, the DataFrame `unionByName` method resolves columns by name. This is equivalent to UNION ALL in SQL. Neither one will remove duplicates.

Below is a check to see if the two dataframes have a matching schema where `union` would be appropriate

```
mattress_df.schema==size_df.schema
-- If we do get the two schemas to match with a simple select statement, then we can use a union
union_count = mattress_df.select("email").union(size_df.select("email")).count()

mattress_count = mattress_df.count()
size_count = size_df.count()

mattress_count + size_count == union_count
```

Non-aggregate and Miscellaneous Functions

Here are a few additional non-aggregate and miscellaneous built-in functions.

Method	Description
col / column	Returns a Column based on the given column name.
lit	Creates a Column of literal value

Method	Description
isnull	Return true iff the column is null
rand	Generate a random column with independent and identically distributed (i.i.d.) samples uniformly distributed in [0.0, 1.0)

```
-- We could select a particular column using the col function
gmail_accounts = sales_df.filter(col("email").contains("gmail"))
display(gmail_accounts)

-- lit can be used to create a column out of a value, which is useful for appending columns.
display(gmail_accounts.select("email", lit(True).alias("gmail user")))
```

DataFrameNaFunctions

DataFrameNaFunctions is a DataFrame submodule with methods for handling null values. Obtain an instance of DataFrameNaFunctions by accessing the `na` attribute of a DataFrame.

Method	Description
drop	Returns a new DataFrame omitting rows with any, all, or a specified number of null values, considering an optional subset of columns
fill	Replace null values with the specified value for an optional subset of columns
replace	Returns a new DataFrame replacing a value with another value, considering an optional subset of columns

```
-- Here we'll see the row count before and after dropping rows with null/NA values.

print(sales_df.count())
print(sales_df.na.drop().count())

-- Since the row counts are the same, we have the no null columns. We'll need to explode items to find some nulls in columns such as items.

sales_exploded_df = sales_df.withColumn("items", explode(col("items")))
display(sales_exploded_df.select("items.coupon"))
print(sales_exploded_df.select("items.coupon").count())
print(sales_exploded_df.select("items.coupon").na.drop().count())

-- We can fill in the missing coupon codes with na.fill

display(sales_exploded_df.select("items.coupon").na.fill("NO COUPON"))
```

Joining DataFrames

The DataFrame `join` method joins two DataFrames based on a given join expression.

Several different types of joins are supported:

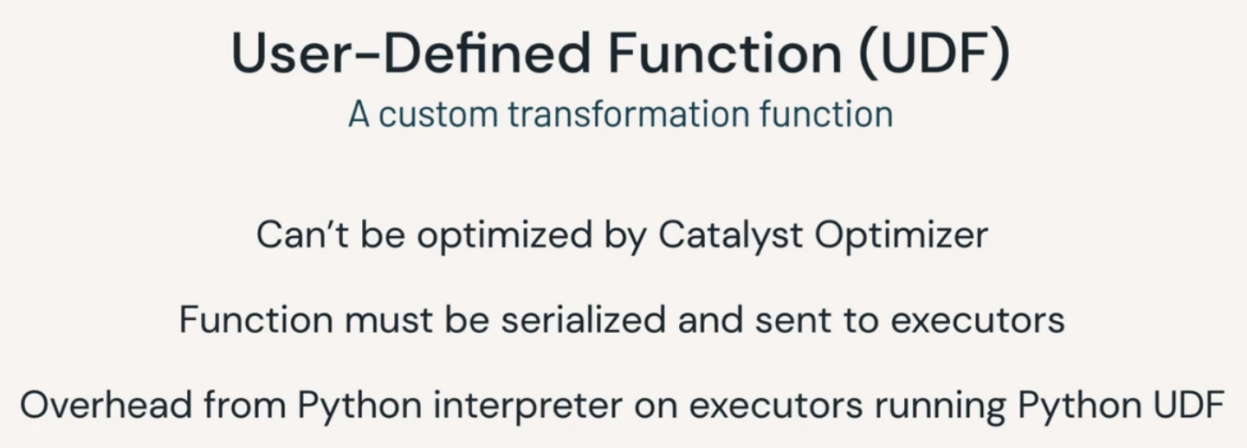
Inner join based on equal values of a shared column called "name" (i.e., an equi join) `df1.join(df2, "name")`

Inner join based on equal values of the shared columns called "name" and "age" `df1.join(df2, ["name", "age"])`

Full outer join based on equal values of a shared column called "name" `df1.join(df2, "name", "outer")`

Left outer join based on an explicit column expression `df1.join(df2, df1["customer_name"] == df2["account_name"], "left_outer")`

```
-- We'll load in our users data to join with our gmail_accounts from above.

users_df = spark.read.format("delta").load(users_path)
display(users_df)

joined_df = gmail_accounts.join(other=users_df, on='email', how = "inner")
display(joined_df)
```



User-Defined Function (UDF)

A custom column transformation function

- Can’t be optimized by Catalyst Optimizer
- Function is serialized and sent to executors
- Row data is deserialized from Spark's native binary format to pass to the UDF, and the results are serialized back into Spark's native format
- For Python UDFs, additional interprocess communication overhead between the executor and a Python interpreter running on each worker node

```
sales_df = spark.read.format("delta").load(sales_path)
display(sales_df)

-- Define a function
-- Define a function (on the driver) to get the first letter of a string from the email field.

def first_letter_function(email):
    return email[0]

first_letter_function("annagray@kaufman.com")

-- Create and apply UDF
-- Register the function as a UDF. This serializes the function and sends it to executors to be able to transform DataFrame records.
first_letter_udf = udf(first_letter_function)
-- Apply the UDF on the email column.

from pyspark.sql.functions import col
display(sales_df.select(first_letter_udf(col("email"))))

-- Register UDF to use in SQL
-- Register the UDF using spark.udf.register to also make it available for use in the SQL namespace.

sales_df.createOrReplaceTempView("sales")
first_letter_udf = spark.udf.register("sql_udf", first_letter_function)

# You can still apply the UDF from Python
display(sales_df.select(first_letter_udf(col("email"))))

%sql
-- You can now also apply the UDF from SQL
SELECT sql_udf(email) AS first_letter FROM sales
```

Use Decorator Syntax (Python Only)

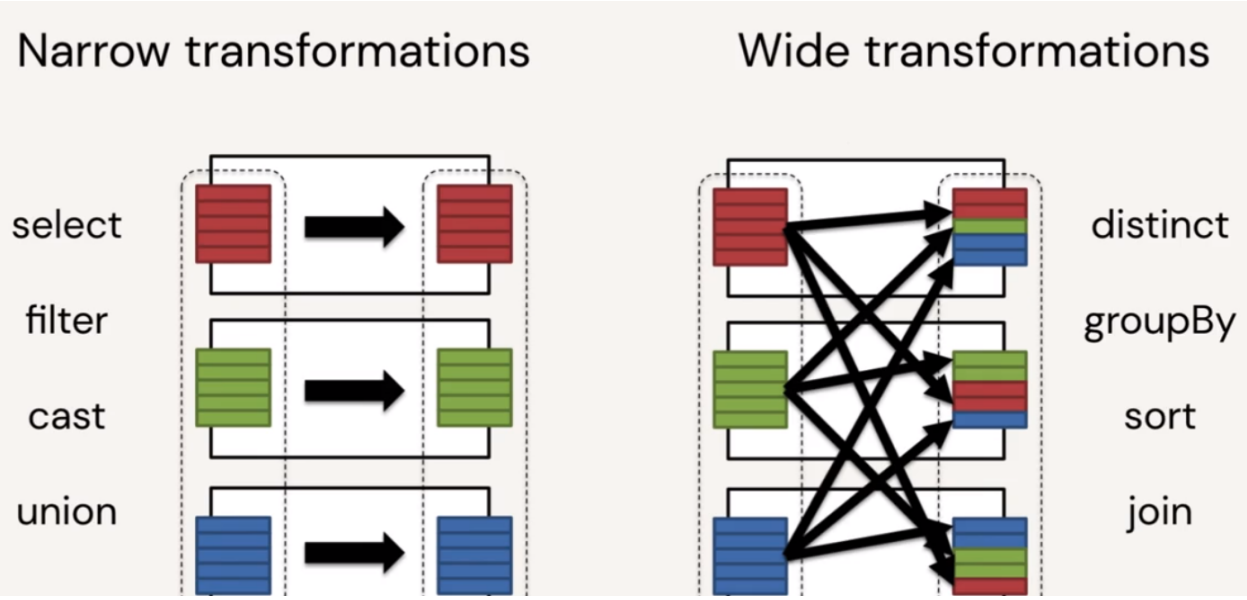
Alternatively, you can define and register a UDF using Python decorator syntax. The `@udf` decorator parameter is the Column datatype the function returns.

You will no longer be able to call the local Python function (i.e., `first_letter_udf("annagray@kaufman.com")` will not work).

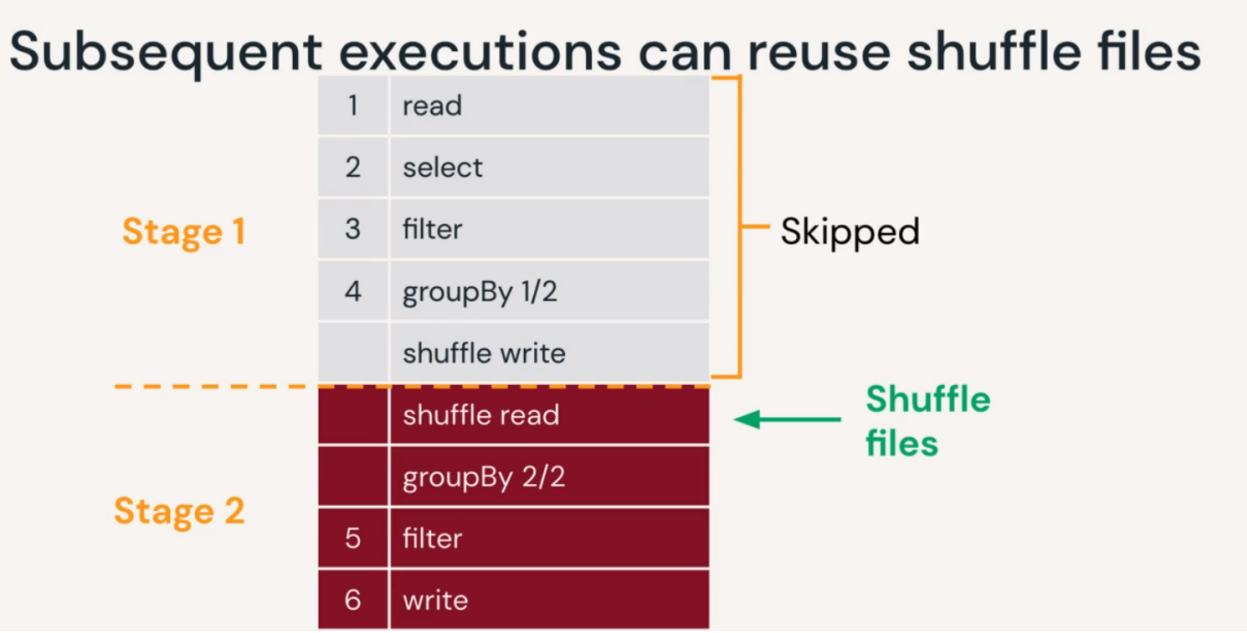
This example also uses Python type hints, which were introduced in Python 3.5. Type hints are not required for this example, but instead serve as "documentation" to help developers use the function correctly. They are used in this example to emphasize that the UDF processes one record at a time, taking a single `str` argument and returning a `str` value.

```
# Our input/output is a string
@udf("string")
def first_letter_udf(email: str) -> str:
    return email[0]

-- And let's use our decorator UDF here.
from pyspark.sql.functions import col

sales_df = spark.read.format("delta").load(f"{datasets_dir}/sales/sales.delta")
display(sales_df.select(first_letter_udf(col("email"))))

-- Pandas/Vectorized UDFs
-- Pandas UDFs are available in Python to improve the efficiency of UDFs. Pandas UDFs utilize Apache Arrow to speed up computation.
-- The user-defined functions are executed using:
Apache Arrow, an in-memory columnar data format that is used in Spark to efficiently transfer data between JVM and Python processes with ne
Pandas inside the function, to work with Pandas instances and APIs
-- As of Spark 3.0, you should always define your Pandas UDF using Python type hints.

import pandas as pd
from pyspark.sql.functions import pandas_udf

# We have a string input/output
@pandas_udf("string")
def vectorized_udf(email: pd.Series) -> pd.Series:
    return email.str[0]

# Alternatively
# def vectorized_udf(email: pd.Series) -> pd.Series:
#     return email.str[0]
# vectorized_udf = pandas_udf(vectorized_udf, "string")

display(sales_df.select(vectorized_udf(col("email"))))

-- We can also register these Pandas UDFs to the SQL namespace.

spark.udf.register("sql_vectorized_udf", vectorized_udf)

%sql
-- Use the Pandas UDF from SQL
SELECT sql_vectorized_udf(email) AS firstLetter FROM sales
```

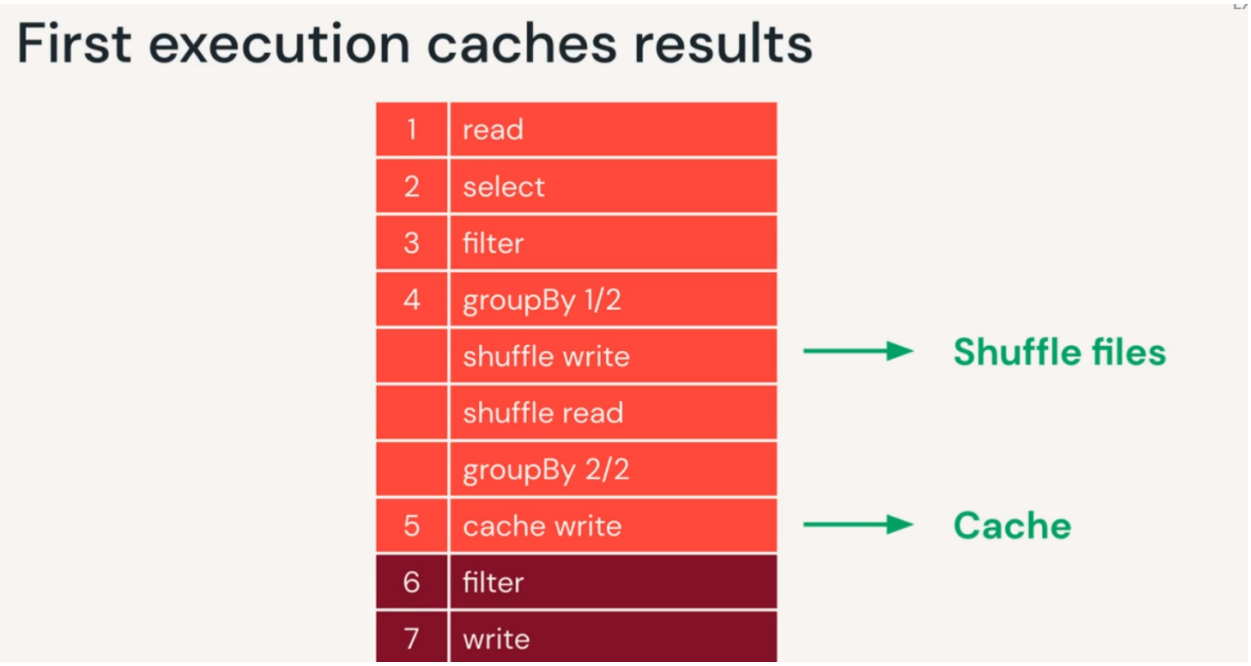
```
Solutions:

-- ASP 3.1L - Revenue by Traffic Lab
# ANSWER
chain_df = (df
            .groupBy("traffic_source")
            .agg(sum(col("revenue")).alias("total_rev"),
                 avg(col("revenue")).alias("avg_rev"))
            .sort(col("total_rev").desc())
            .limit(3)
            .withColumn("avg_rev", round("avg_rev", 2))
            .withColumn("total_rev", round("total_rev", 2))
           )

display(chain_df)
```

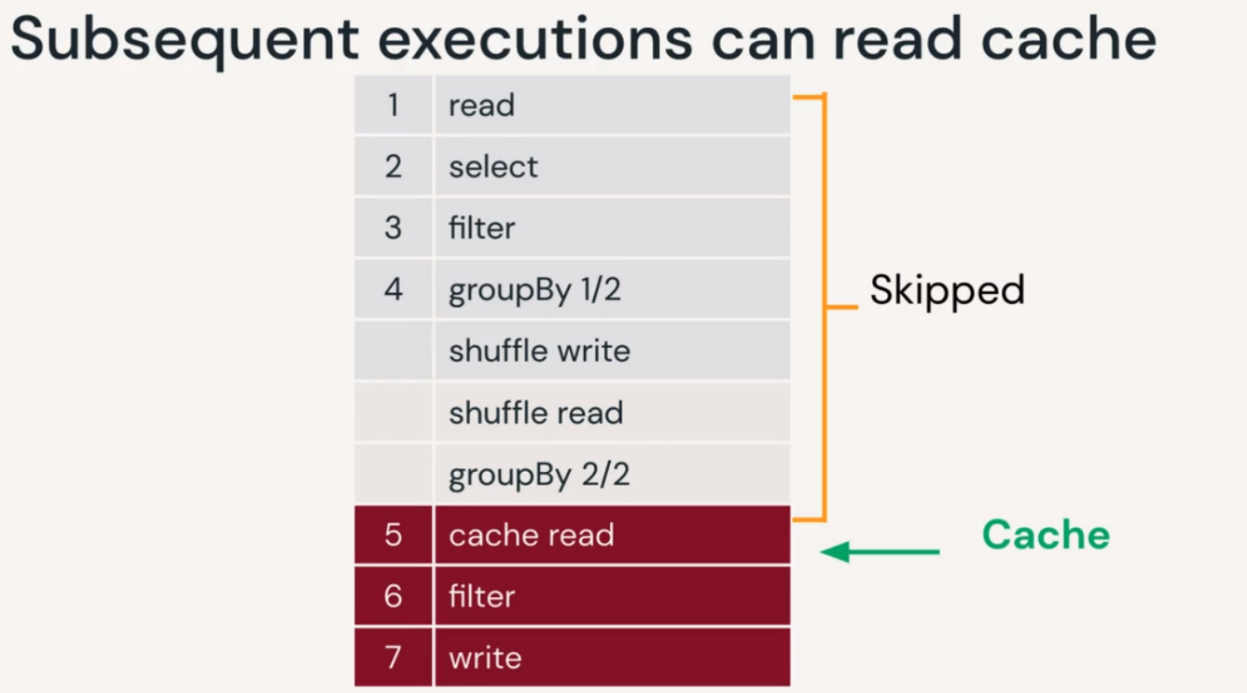


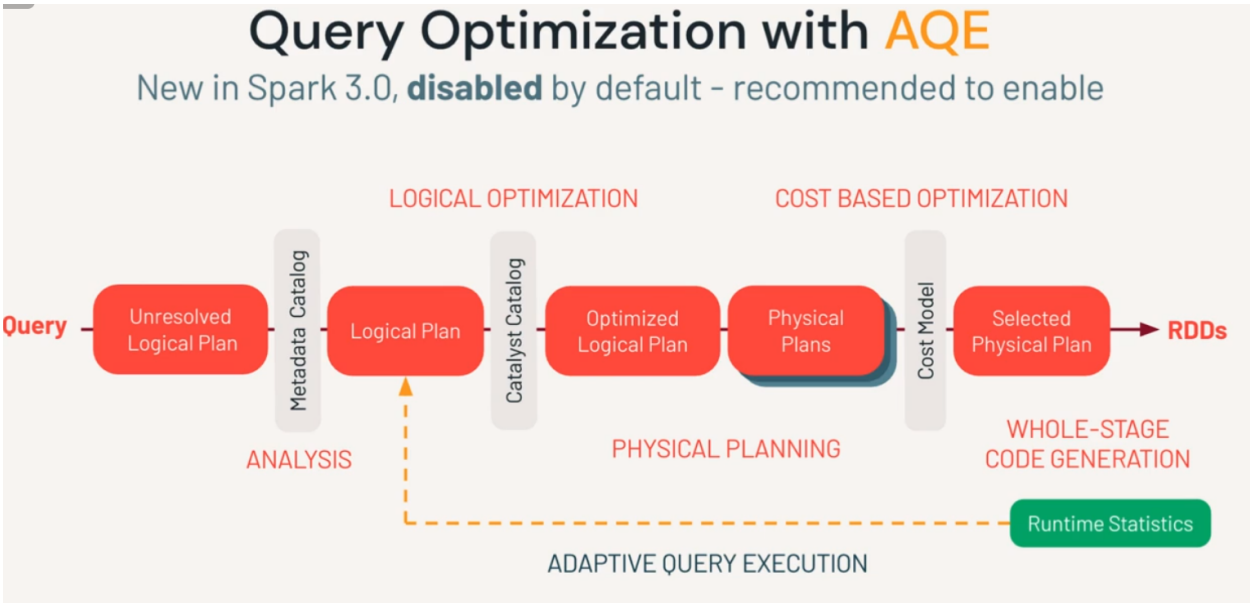
Shuffles introduce stage boundaries. Shuffles demarcate stage boundaries. First exection creates shuffle files.



Cache data: explicitly accomplish the same thing.







AQE: New feature of 3.0, create run time statistics. Based on statistics of finished planned nodes and reoptimize the exection plans of the remaining queries

Query Optimization

We'll explore query plans and optimizations for several examples including logical optimizations and exanples with and without predicate pushdown.

Objectives

1. Logical optimizations

2. Predicate pushdown
3. No predicate pushdown

Methods

- DataFrame: `explain`

Logical Optimization

`explain(..)` prints the query plans, optionally formatted by a given explain mode. Compare the following logical plan & physical plan, noting how Catalyst handled the multiple `filter` transformations.

```
-- This is bad

from pyspark.sql.functions import col

limit_events_df = (df
                   .filter(col("event_name") != "reviews")
                   .filter(col("event_name") != "checkout")
                   .filter(col("event_name") != "register")
                   .filter(col("event_name") != "email_coupon")
                   .filter(col("event_name") != "cc_info")
                   .filter(col("event_name") != "delivery")
                   .filter(col("event_name") != "shipping_info")
                   .filter(col("event_name") != "press")
                  )

limit_events_df.explain(True)

Of course, we could have written the query originally using a single filter condition ourselves.
Compare the previous and following query plans. (This is better)

better_df = (df
             .filter((col("event_name").isNotNull()) &
                     (col("event_name") != "reviews") &
                     (col("event_name") != "checkout") &
                     (col("event_name") != "register") &
                     (col("event_name") != "email_coupon") &
                     (col("event_name") != "cc_info") &
                     (col("event_name") != "delivery") &
                     (col("event_name") != "shipping_info") &
                     (col("event_name") != "press"))
            )

better_df.explain(True)
```

Caching

By default the data of a DataFrame is present on a Spark cluster only while it is being processed during a query -- it is not automatically persisted on the cluster afterwards. (Spark is a data processing engine, not a data storage system.) You can explicity request Spark to persist a DataFrame on the cluster by invoking its `cache` method.

If you do cache a DataFrame, you should always explictly evict it from cache by invoking `unpersist` when you no longer need it.

Caching a DataFrame can be appropriate if you are certain that you will use the same DataFrame multiple times, as in:

- Exploratory data analysis
- Machine learning model training

Aside from those use cases, you should **not** cache DataFrames because it is likely that you'll *degrade* the performance of your application.

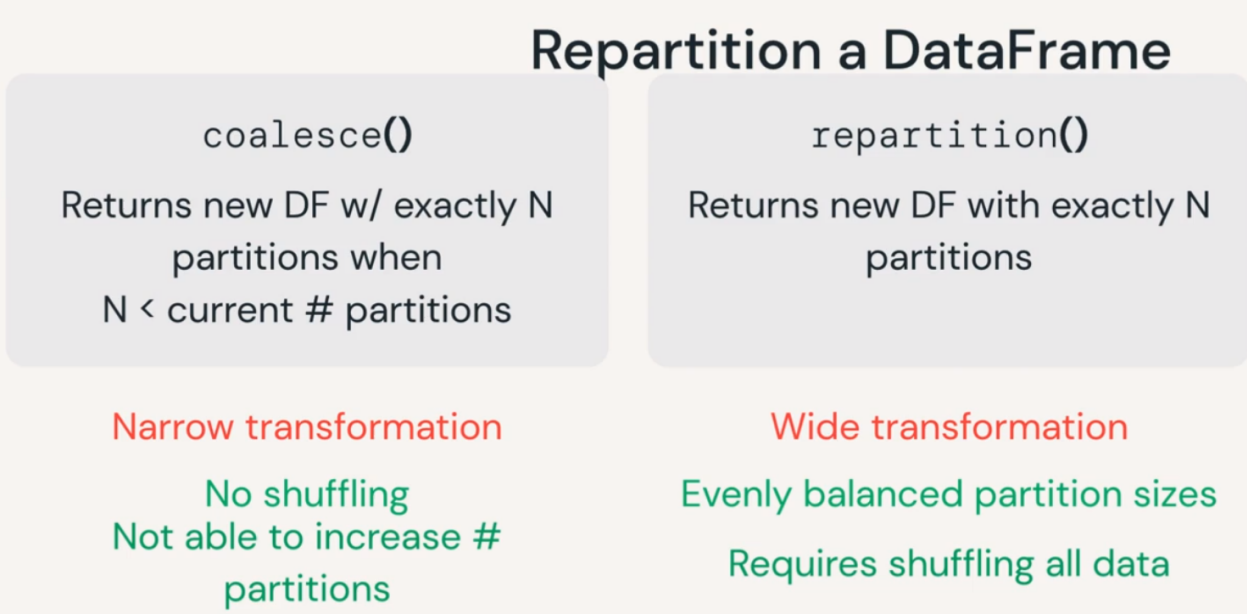
- Caching consumes cluster resources that could otherwise be used for task execution
- Caching can prevent Spark from performing query optimizations, as shown in the next example

```
Predicate Pushdown
Here is example reading from a JDBC source, where Catalyst determines that predicate pushdown can take place.

jdbc_url = "jdbc:postgresql://54.213.33.240/training"
```

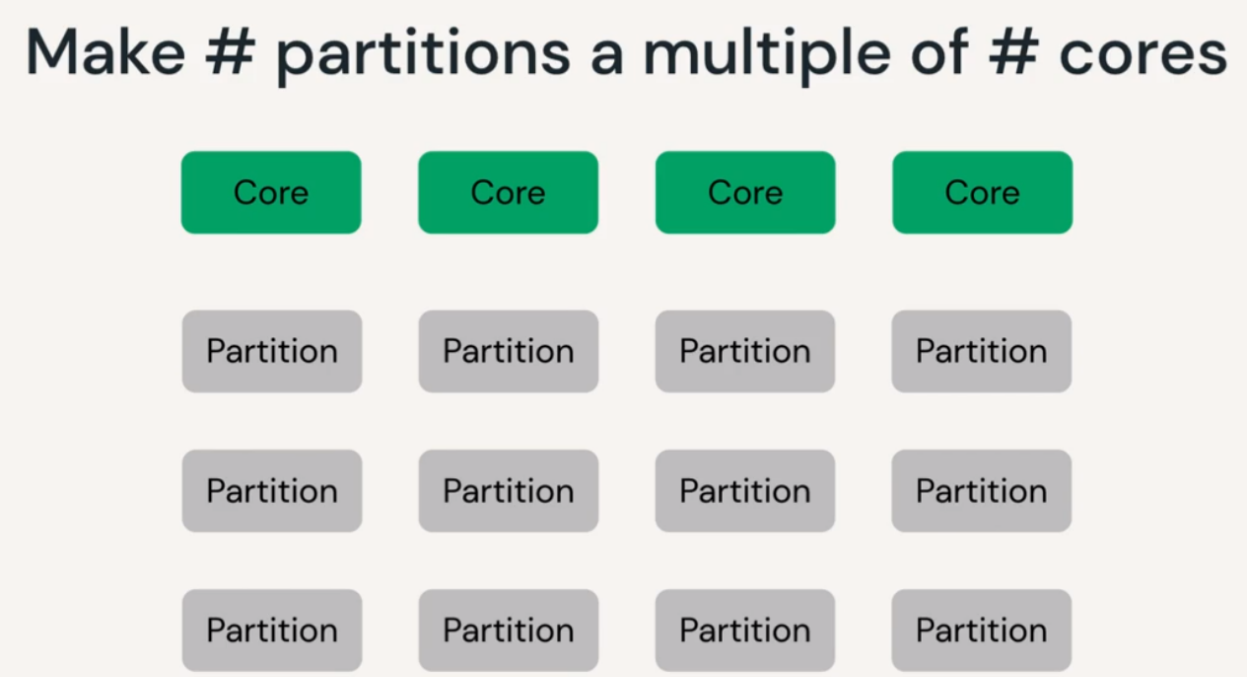
```
# Username and Password w/read-only rights
conn_properties = {
    "user" : "training",
    "password" : "training"
}

pp_df = (spark
         .read
         .jdbc(url=jdbc_url,                # the JDBC URL
               table="training.people_1m",   # the name of the table
               column="id",                  # the name of a column of an integral type that will be used for partitioning
               lowerBound=1,                 # the minimum value of columnName used to decide partition stride
               upperBound=1000000,           # the maximum value of columnName used to decide partition stride
               numPartitions=8,              # the number of partitions/connections
               properties=conn_properties    # the connection properties
              )
         .filter(col("gender") == "M")   # Filter the data by gender
        )

pp_df.explain(True)

Note the lack of a Filter and the presence of a PushedFilters in the Scan.
The filter operation is pushed to the database and only the matching records are sent to Spark.
This can greatly reduce the amount of data that Spark needs to ingest.

No Predicate Pushdown
In comparison, caching the data before filtering eliminates the possibility for the predicate push down.

cached_df = (spark
            .read
            .jdbc(url=jdbc_url,
                  table="training.people_1m",
                  column="id",
                  lowerBound=1,
                  upperBound=1000000,
                  numPartitions=8,
                  properties=conn_properties
                 )
            )

cached_df.cache()
filtered_df = cached_df.filter(col("gender") == "M")

filtered_df.explain(True)

In addition to the Scan (the JDBC read) we saw in the previous example, here we also see the InMemoryTableScan followed by a Filter in the

This means Spark had to read ALL the data from the database and cache it, and then scan it in cache to find the records matching the filter
```

Partition is a small piece of data

```
Get # cores in cluster
Core = Thread available for parallel execution

sc.defaultParallelism
spark.sparkContext.defaultParallelism

Get # paritions of data
df.rdd.getNumPartitions()
```



coalesce is faster than repartition



Goal is to make sure that every slots and cores are used. Don't underultilise.

If i read my data and it comes down as 10 paritions. Should i increase partitions to 16 or reduce to 8. Ans: Look into size of paritions and see if it exceeds 200mb. If exceeds, increase to 16 paritions. If below, decrease to 8.

Default shuffle partitions

```
spark.conf.get("spark.sql.shuffle.partitions")
```

```
spark.conf.set("spark.sql.shuffle.partitions", "8")
```

Partitioning Guidelines

- Err on the side of too many small than too few large partitions
- Don’t allow partition size to increase > 200MB per 8GB of core total memory
- Size default shuffle partitions by dividing largest shuffle stage input by the target partition size

e.g. 4TB / 200MB = 20,000 shuffle partition count

3.0 AQE : Dynamically coalesce shuffle partitions

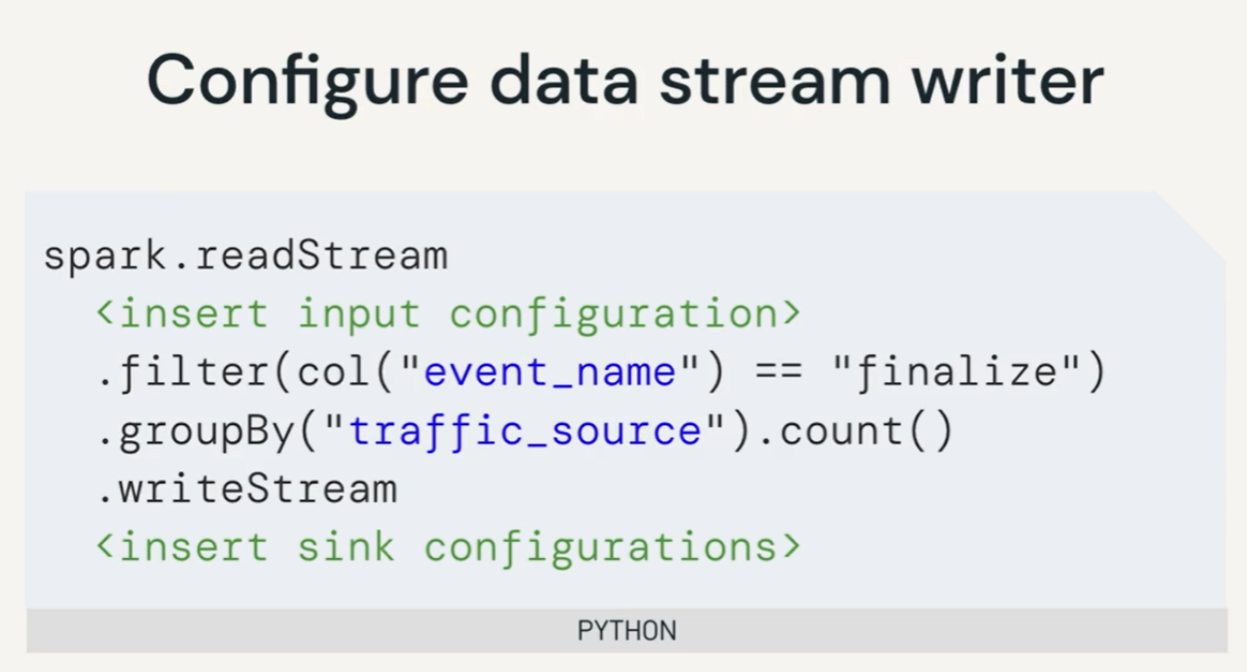
```
Access SparkContext through SparkSession to get the number of cores or slots.
Use the defaultParallelism attribute to get the number of cores in a cluster.

print(spark.sparkContext.defaultParallelism) -> Results: 2

SparkContext is also provided in Databricks notebooks as the variable sc.

print(sc.defaultParallelism)

--repartition
Returns a new DataFrame that has exactly n partitions.

Wide transformation
Pro: Evenly balances partition sizes
Con: Requires shuffling all data

repartitioned_df = df.repartition(8)
repartitioned_df.rdd.getNumPartitions()


--coalesce
Returns a new DataFrame that has exactly n partitions, when fewer partitions are requested.

If a larger number of partitions is requested, it will stay at the current number of partitions.

Narrow transformation, some partitions are effectively concatenated
Pro: Requires no shuffling
Cons:
```

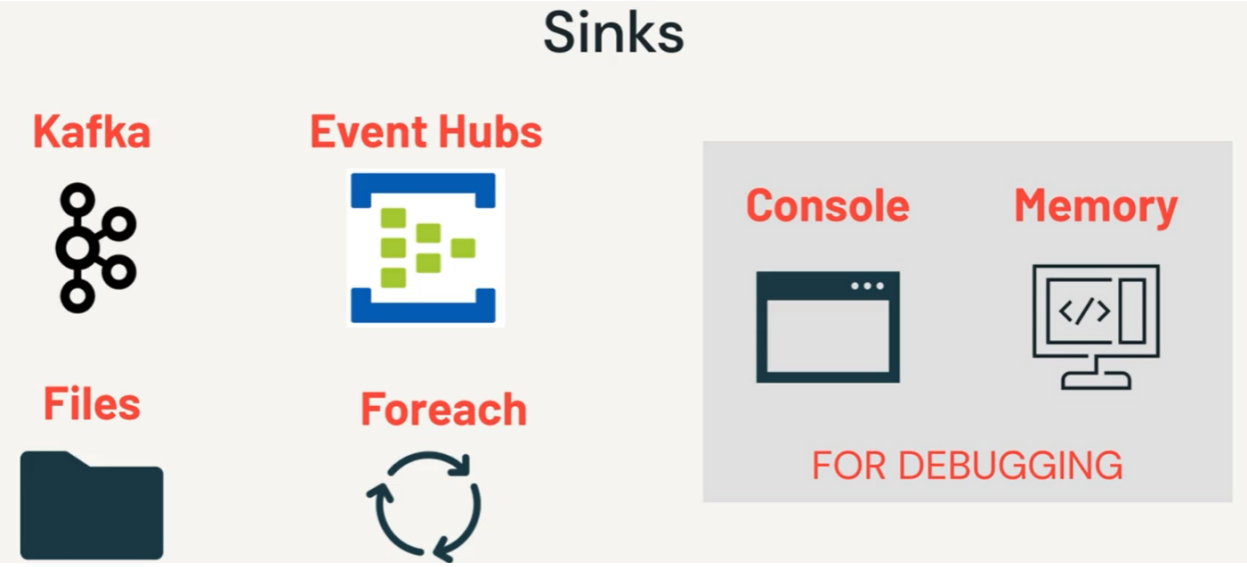
```
Is not able to increase # partitions
Can result in uneven partition sizes

coalesce_df = df.coalesce(8)
coalesce_df.rdd.getNumPartitions() -> Results: 4

Configure default shuffle partitions
Use the SparkSession's conf attribute to get and set dynamic Spark configuration properties.
The spark.sql.shuffle.partitions property determines the number of partitions that result from a shuffle.
Let's check its default value:

spark.conf.get("spark.sql.shuffle.partitions") -> Results: 200
Assuming that the data set isn't too large, you could configure the default number of shuffle partitions to match the number of cores:
spark.conf.set("spark.sql.shuffle.partitions", spark.sparkContext.defaultParallelism)
print(spark.conf.get("spark.sql.shuffle.partitions")) -> Results :2

In Spark 3, AQE is now able to dynamically coalesce shuffle partitions at runtime.
This means that you can set spark.sql.shuffle.partitions based on the largest data set your application processes and
allow AQE to reduce the number of partitions automatically when there is less data to process.

The spark.sql.adaptive.enabled configuration option controls whether AQE is turned on/off.

spark.conf.get("spark.sql.adaptive.enabled")
```

```
ASP 4.2L

# ANSWER

source_file = f"{datasets_dir}/people/people-with-dups.txt"
delta_dest_dir = working_dir + "/people"

# In case it already exists
dbutils.fs.rm(delta_dest_dir, True)

# dropDuplicates() will introduce a shuffle, so it helps to reduce the number of post-shuffle partitions.
spark.conf.set("spark.sql.shuffle.partitions", 8)

# Okay, now we can read this thing
df = (spark
      .read
      .option("header", "true")
      .option("inferSchema", "true")
      .option("sep", ":")
      .csv(source_file)
     )

# ANSWER
from pyspark.sql.functions import col, lower, translate

deduped_df = (df
              .select(col("*"),
                      lower(col("firstName")).alias("lcFirstName"),
                      lower(col("lastName")).alias("lcLastName"),
                      lower(col("middleName")).alias("lcMiddleName"),
                      translate(col("ssn"), "-", "").alias("ssnNums")
                      # regexp_replace(col("ssn"), "-", "").alias("ssnNums")  # An alternate function to strip the hyphens
                      # regexp_replace(col("ssn"), """^(\d{3})(\d{2})(\d{4})$""", "$1-$2-$3").alias("ssnNums")  # An alternate that adds hyp
                     )
              .dropDuplicates(["lcFirstName", "lcMiddleName", "lcLastName", "ssnNums", "gender", "birthDate", "salary"])
              .drop("lcFirstName", "lcMiddleName", "lcLastName", "ssnNums")
             )

# ANSWER

# Now, write the results in Delta format as a single file. We'll also display the Delta files to make sure they were written as expected.

(deduped_df
 .repartition(1)
 .write
 .mode("overwrite")
 .format("delta")
 .save(delta_dest_dir)
)

display(dbutils.fs.ls(delta_dest_dir))

verify_files = dbutils.fs.ls(delta_dest_dir)
verify_delta_format = False
```

```
verify_num_data_files = 0
for f in verify_files:
    if f.name == "_delta_log/":
        verify_delta_format = True
    elif f.name.endswith(".parquet"):
        verify_num_data_files += 1

assert verify_delta_format, "Data not written in Delta format"
assert verify_num_data_files == 1, "Expected 1 data file written"

verify_record_count = spark.read.format("delta").load(delta_dest_dir).count()
assert verify_record_count == 100000, "Expected 100000 records in final result"

del verify_files, verify_delta_format, verify_num_data_files, verify_record_count
print("All test pass")
```

Structured Streaming:





Trigger Types

Default	Process each micro-batch as soon as the previous one has been processed
Fixed interval	Micro-batch processing kicked off at the user-specified interval
One-time	Process all of the available data as a single micro-batch and then automatically stop the query
Continuous Processing	Long-running tasks that continuously read, process, and write data as soon events are available *experimental, Spark 2.3+*

End-to-end fault tolerance

Guaranteed in Structured Streaming by

Checkpointing and write-ahead logs

Idempotent sinks

Replayable data sources

Execute to create a streaming query

```python
spark.readStream
  <insert input configuration>
  .filter(col("event_name") == "finalize")
  .groupBy("traffic_source").count()
  .writeStream
  <insert sink configurations>
  .start()
```

PYTHON

Streaming Query Operations

- Stop stream
- Await termination
- Status
- Is active
- Recent progress
- Name, ID, runID

Complete streaming query

```
spark.readStream
  .schema(dataSchema)
  .option("maxFilesPerTrigger", 1)
  .parquet(eventsPath)
  .filter(col("event_name") == "finalize")
  .groupBy("traffic_source").count()
  .writeStream
  .outputMode("append")
  .format("parquet")
  .queryName("program_ratings")
  .trigger(processingTime="3 seconds")
  .option("checkpointLocation", checkpointPath)
  .start(outputPathDir)
```

```
-- Build streaming DataFrames
-- Obtain an initial streaming DataFrame from a Delta-format file source

df = (spark
      .readStream
      .option("maxFilesPerTrigger", 1)
      .format("delta")
      .load(events_path)
     )

df.isStreaming

-- Apply some transformations, producing new streaming DataFrames.

from pyspark.sql.functions import col, approx_count_distinct, count

email_traffic_df = (df
                    .filter(col("traffic_source") == "email")
                    .withColumn("mobile", col("device").isin(["iOS", "Android"]))
                    .select("user_id", "event_timestamp", "mobile")
                   )

email_traffic_df.isStreaming

-- Write streaming query results
-- Take the final streaming DataFrame (our result table) and write it to a file sink in "append" mode.

checkpoint_path = f"{working_dir}/email_traffic/checkpoint"
output_path = f"{working_dir}/email_traffic/output"

devices_query = (email_traffic_df
                 .writeStream
                 .outputMode("append")
                 .format("delta")
                 .queryName("email_traffic")
                 .trigger(processingTime="1 second")
                 .option("checkpointLocation", checkpoint_path)
                 .start(output_path)
                )
```

```
-- Monitor streaming query
-- Use the streaming query "handle" to monitor and control it.

devices_query.id
devices_query.status
devices_query.lastProgress

import time
# Run for 10 more seconds
time.sleep(10)

devices_query.stop()

devices_query.awaitTermination()
```

Checkpoints

A checkpoint helps build fault-tolerant and resilient Spark applications. In Spark Structured Streaming, it maintains intermediate state on HDFS compatible file systems to recover from failures. To specify the checkpoint in a streaming query, we use the `checkpointLocation` parameter. The parameter `"checkpointLocation”` enables the checkpoint and specifies the location where we keep checkpoint information.

https://medium.com/expedia-group-tech/apache-spark-structured-streaming-checkpoints-and-triggers-4-of-6-b6f15d5cfd8d#:~:text=A checkpoint helps build fault,we use the checkpointLocation parameter.

Practice exams

Sample Pyspark Script

General Spark