

# CSP554

# BIG DATA TECHNOLOGIES

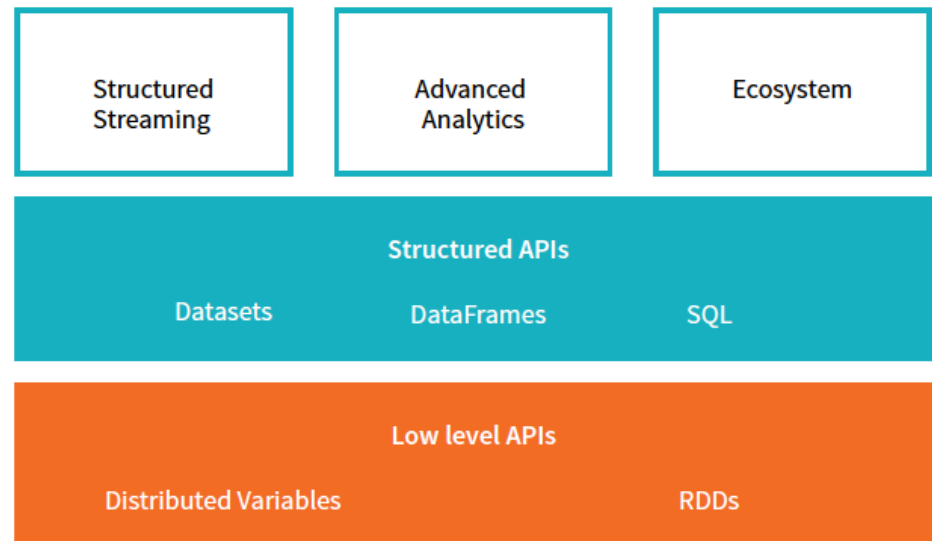
---

Module 07

Spark SQL (DataFrames)

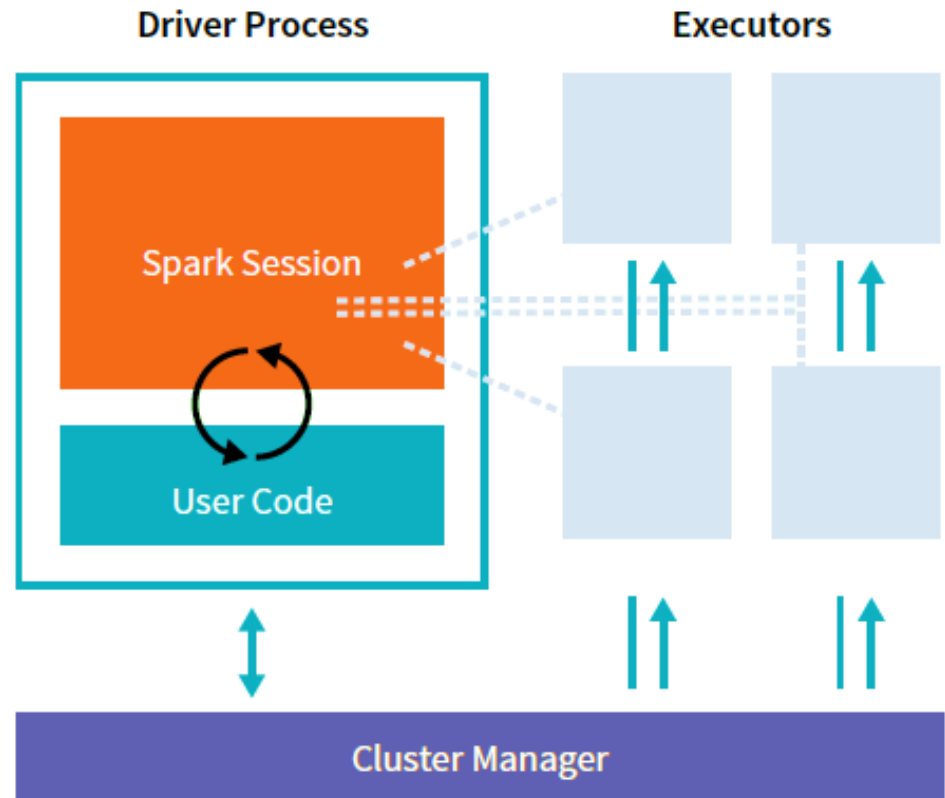
# What is Apache Spark?

- Apache Spark is a unified computing engine and a set of libraries for parallel data processing on computer clusters
- Spark is the most actively developed open source engine for this task; making it the de facto tool for any developer or data scientist interested in big data.
- Spark supports multiple widely used programming languages (Python, Java, Scala and R), includes libraries for diverse tasks ranging from SQL to streaming and machine learning



# Spark's Basic Architecture

- Spark applications consist of a driver process and a set of executor processes
- The driver process is responsible for distributing our program's commands across the executors in order to complete our task.
- The executors are responsible for actually executing the work that the driver assigns them
- The cluster manager (YARN) controls physical (or virtual) machines and allocates resources to Spark Applications



# Spark Data Abstractions

- Spark has several core abstractions
  - Datasets
  - DataFrames
  - SQL Tables
  - Resilient Distributed Datasets (RDDs)
- These abstractions all represent distributed collections of data however they have different interfaces for working with that data
- The easiest and most efficient are DataFrames and SQL Tables

# Spark Structured APIs

- A tool for manipulating all sorts of data, from unstructured log files to semi-structured CSV files and highly structured Parquet files
- Refers to three core types of distributed collection APIs:
  - Datasets
  - DataFrames
  - SQL (like) tables
- The majority of the Structured APIs apply to both batch and streaming computation.
- This means that when you work with the Structured APIs, it should be simple to migrate from batch to streaming (or vice versa) with little effort

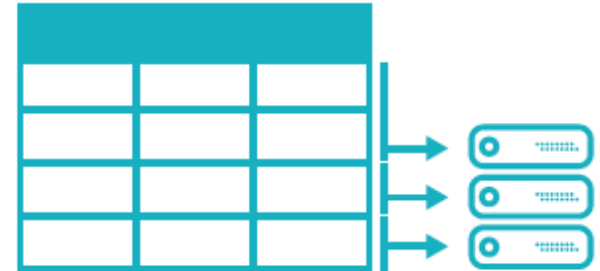
# DataFrames: A Quick Overview

- 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 are the DataFrame schema
- A simple analogy would be a spreadsheet with named columns.
- The fundamental difference is that while spreadsheet data sits on one computer in one specific location, a Spark DataFrame represents data that can span thousands of computers

Spreadsheet on a single machine



Table or DataFrame partitioned across servers in a data center



# DataFrame

- 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 the schema
- 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, a Spark DataFrame can span thousands of computers
- Each record in a DataFrame must be of type Row
- The DataFrame itself can be considered as contained an Array where each entry is of type Row

Spreadsheet on a single machine



Table or DataFrame partitioned across servers in a data center



# 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 DataFrame or DataSet rows that sit on one physical (or virtual) machine in our cluster
- A DataFrame's partitions represent how the data is distributed across a 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



# DataFrames and Datasets

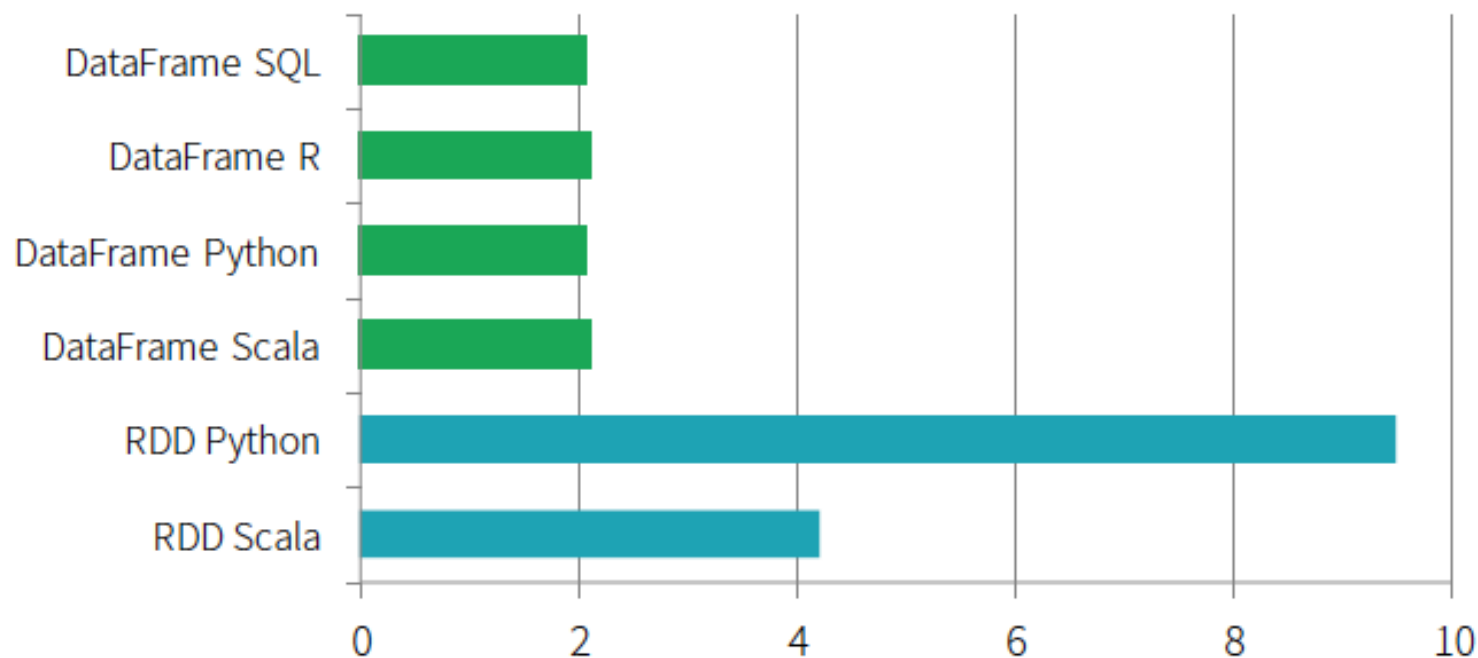
- DataFrames and Datasets are (distributed) table-like collections with well-defined rows and columns
- Each row must have the same number of columns as all the other rows (although you can use null to specify the absence of a value).
- Each column has type information that must be consistent for every row in the collection
- DataFrames and Datasets represent immutable, lazily evaluated plans that specify what operations to apply to data residing at a location to generate some output
- When we perform an action on a DataFrame, we instruct Spark to perform the actual transformations and return the result
- These represent plans of how to manipulate rows and columns to compute the user's desired result.

# DataFrames Versus Datasets

- Within the Structured APIs, there are two more APIs, the “untyped” DataFrames and the “typed” Datasets
  - To say that DataFrames are untyped is slightly inaccurate; they have types, but Spark maintains them completely and only checks whether those types line up to those specified in the schema at *runtime*
  - Datasets, on the other hand, check whether types conform to the specification at *compile time*
  - Datasets are only available to Java Virtual Machine (JVM)–based languages (Scala and Java)
- For the most part, you’re likely to work with DataFrames.
- DataFrames are simply Datasets of Type Row
  - That is, each record in a DataFrame is an instance of type Row
  - The “Row” type is Spark’s internal representation of its optimized in-memory format for computation

# DataFrame Performance

- DataFrames can be significantly faster than RDDs
- And they perform the same, regardless of language



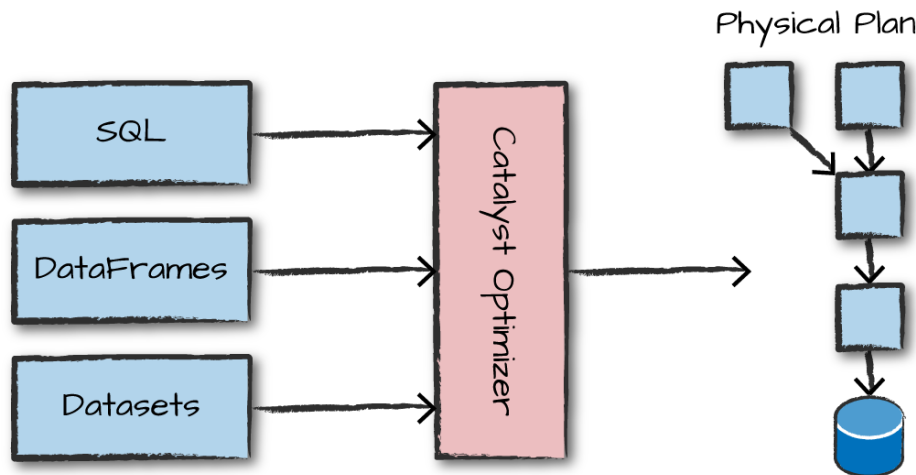
Time to aggregate 10 million integer pairs (in seconds)

# Schemas

- Schemas consist of name and type information for each column of a DataFrame
- You can define schemas manually or have Spark infer the schema from a data source as it is read in to a DataFrame

# Overview of Structured API Execution

1. Write DataFrame/Dataset/SQL Code.
2. If valid code, Spark converts this to a *Logical Plan*
3. Spark transforms this *Logical Plan* to a *Physical Plan*, checking for optimizations along the way
4. Spark then executes this *Physical Plan* (RDD manipulations) on the cluster

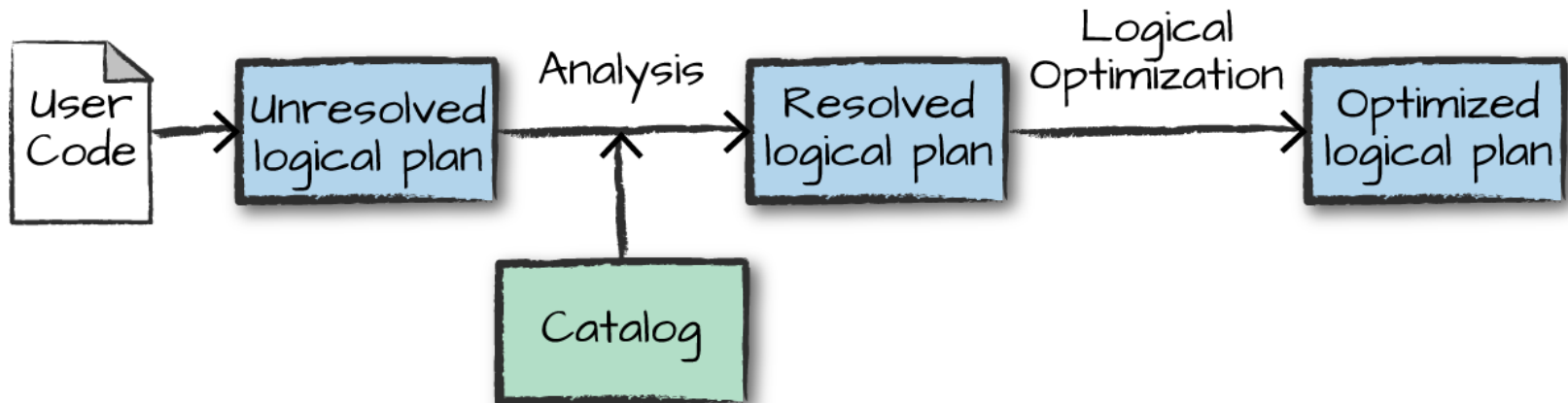


Code is submitted to Spark either through the console or via a submitted job. This code then passes through the Catalyst Optimizer, which decides how the code should be executed and lays out a plan for doing so before, finally, the code is run and the result is returned to the user

# Structured API Execution Details

## Logical Plan

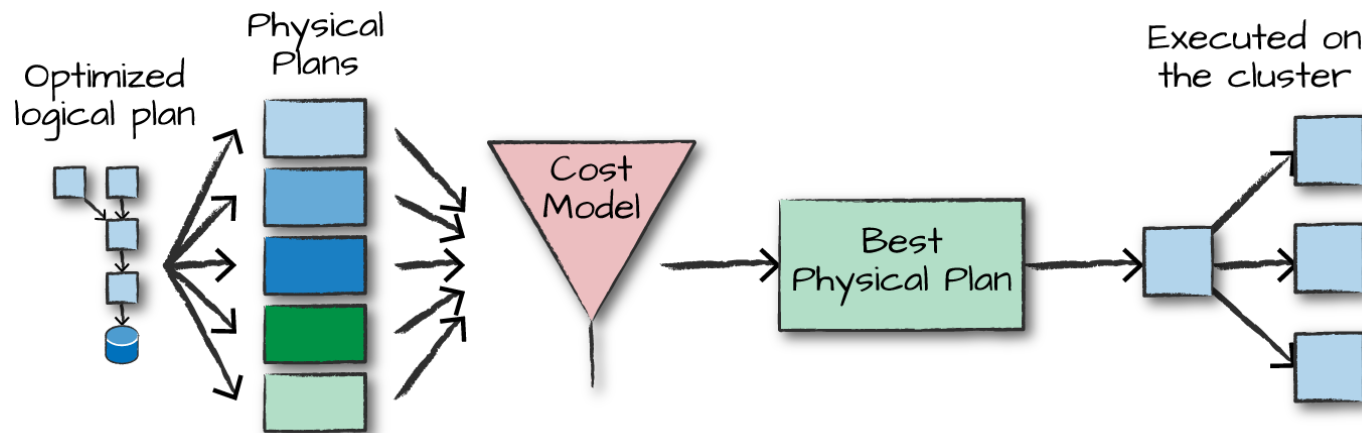
- This logical plan represents a set of abstract transformations to convert the user's set of expressions into the most optimized version
- It does this by converting user code into an *unresolved logical plan*. This plan is unresolved because although your code might be valid, the tables or columns that it refers to might or might not exist.
- Spark uses the *catalog*, a repository of all table and DataFrame information, to *resolve* columns and tables in the *analyzer*.
- The result is passed through the Catalyst Optimizer, a collection of rules that attempt to optimize the logical plan by pushing down predicates or selections.



# Structured API Execution Details

## Physical Plan

- The *physical plan*, often called a Spark plan, specifies how the logical plan will execute on the cluster by generating different physical execution strategies and comparing them through a cost model,
- An example of the cost comparison might be choosing how to perform a given join by looking at the physical attributes of a given table (how big the table is or how big its partitions are)
- Physical planning results in a series of RDDs and transformations
- Spark then runs all of this code on the Hadoop cluster



# An End to End Example

- Here will will work an example step by step using some flight data available here from the United States Bureau of Transportation statistics
- Here is a same of the file including the header row with column labels and a few data rows

```
DEST_COUNTRY_NAME,ORIGIN_COUNTRY_NAME,count
```

```
United States,Romania,15
```

```
United States,Croatia,1
```

```
United States,Ireland,344
```



# An End to End Example

- Spark includes the ability to read and write from a large number of data sources
- In order to read this data in, we will use a `DataFrameReader` that is associated with our `SparkSession`
  - In doing so, we will specify the file format as well as any options we want to specify
  - In our case, we want to do something called schema inference, we want Spark to take the best guess at what the schema of our `DataFrame` should be.
- We also want to specify that the first row is the header in the file, we'll specify that as an option too.
- To get this information Spark will read in a little bit of the data and then attempt to parse the types in those rows according to the types available in Spark

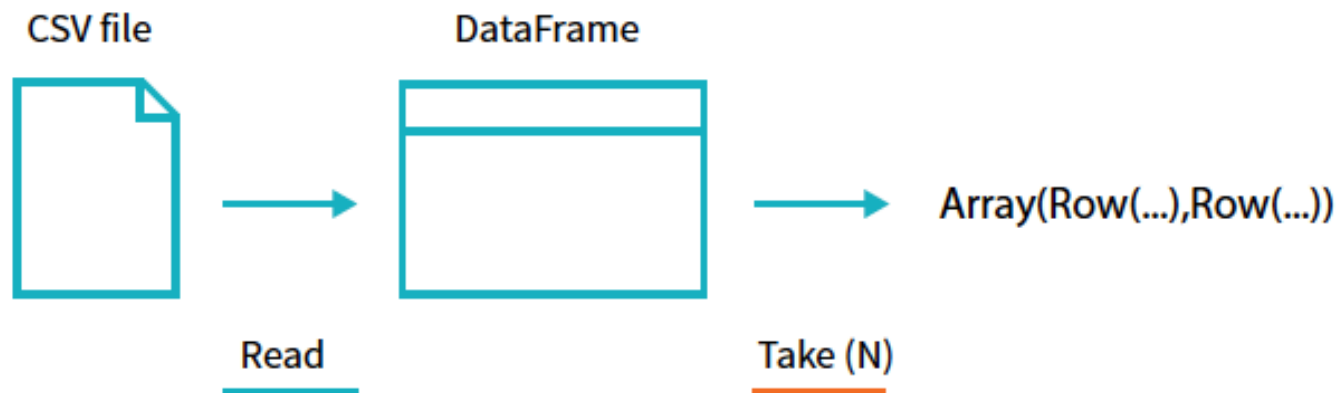
```
flightData2015 = spark\  
.read\  
.option("inferSchema", "true")\  
.option("header", "true")\  
.csv("hdfs:///mnt/defg/flight-data/csv/2015-summary.csv")
```

# An End to End Example

- Each of these DataFrames have a set of columns with an unspecified number of rows
- The reason the number of rows is “unspecified” is because reading data is a transformation, and is therefore a lazy operation
- Spark only peeked at a couple of rows of data to try to guess what types each column should be
- If we perform the take action on the DataFrame, we will be able to see the results

```
flightData2015.take(3)
```

```
Array([United States,Romania,15], [United States,Croatia...
```



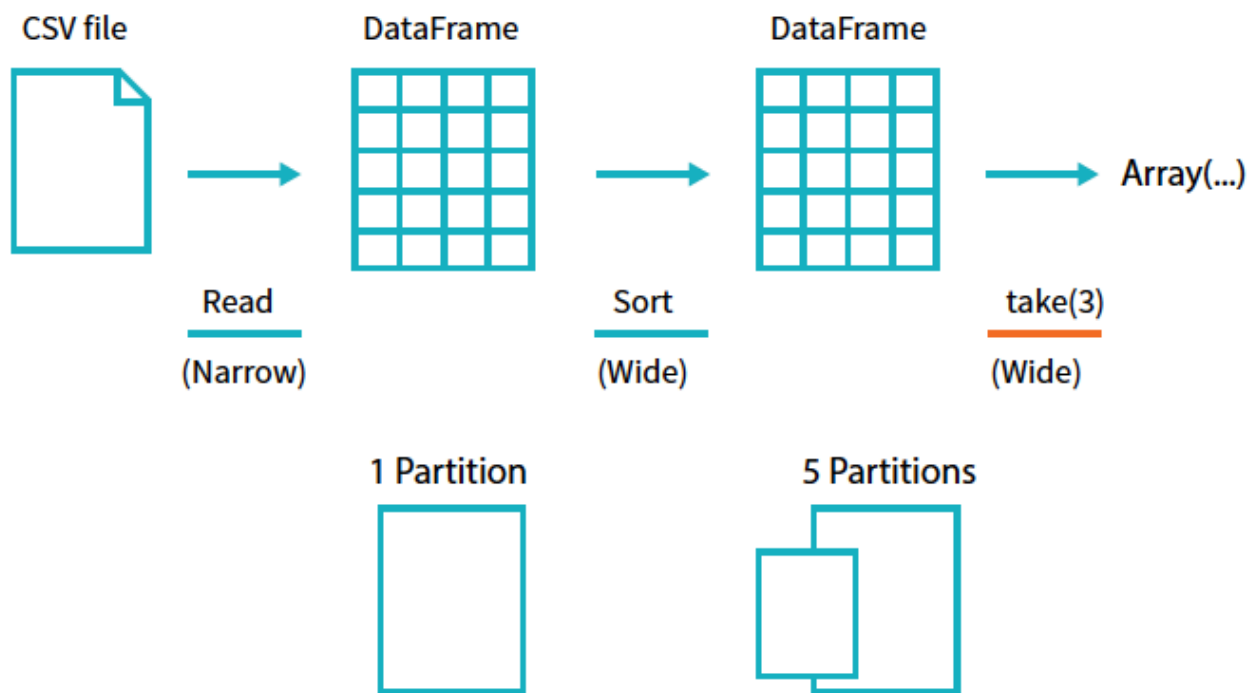
# An End to End Example

- Let's specify some more transformations!
- Now we will sort our data according to the count column which is an integer type
- Nothing happens to the data when we call sort because it's just a transformation
- However, Spark is building up a plan for how it will execute this across the cluster
- Now, just like we did before, we can specify an action in order to kick off this plan
- However before doing that, we're going to set a configuration
  - By default, when we perform a shuffle Spark will output two hundred shuffle partitions
  - We will set this value to five in order to reduce the number of the output partitions from the shuffle

```
spark.conf.set("spark.sql.shuffle.partitions", "5")  
flightData2015.sort("count").take(2)  
... Array([United States,Singapore,1], [Moldova,United States,1])
```

# An End to End Example

- This operation is illustrated in the following image
  - In addition to the logical transformations, we include the physical partition count as well



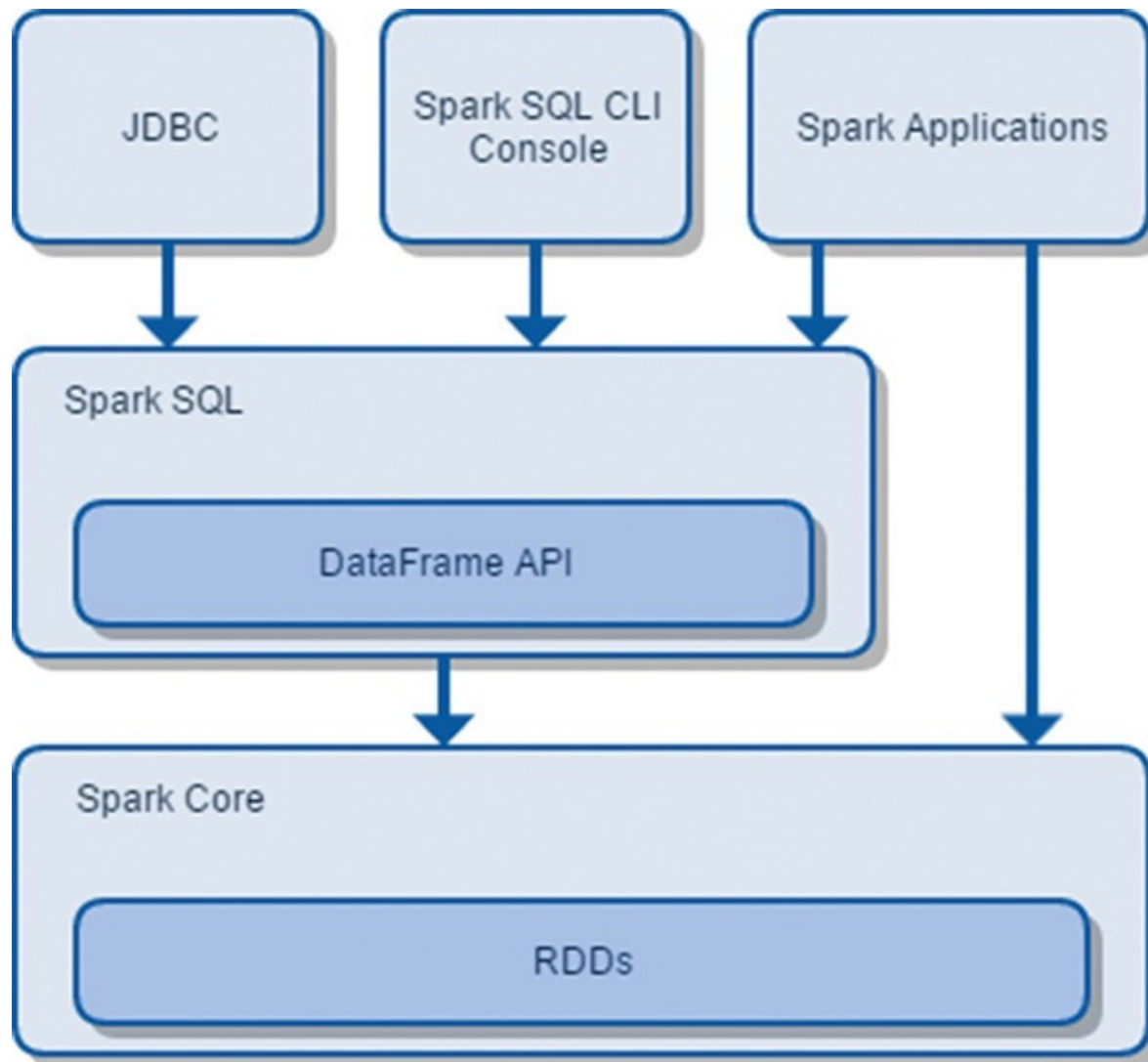
# Spark SQL

- The Hive project was instrumental in providing the pattern of SQL-like access to unstructured data in HDFS...
- But performance and user experience clearly fell well short of database systems
- The batch nature of MapReduce, which is the processing engine behind Hive, was not suited to interactive queries or real-time applications
- Spark SQL provides an SQL abstraction to its RDD-based storage, scheduling, and execution model
- Many key characteristics of core Spark are inherited by Spark SQL
  - Including lazy evaluation and mid-query fault tolerance
- Moreover, Spark SQL can be used in conjunction with the Spark core API within a single application.

# Spark SQL

- One use of Spark SQL is to execute SQL queries written using either a basic SQL syntax or HQL
- Spark SQL can also be used to read data from an existing Hive files
- You interact with the SQL interface using the command line or via JDBC/ODBC
- The whitepaper titled “Spark SQL: Relational Data Processing in Spark” is recommended further reading

# Spark SQL Architecture



# DataFrames and SQL

- Spark SQL allows you to register any DataFrame as a table and query it using pure SQL
- There is no performance difference between writing SQL queries or writing DataFrame code
- They both “compile” to the same underlying plan
- Any DataFrame can be made into a table or view with one simple method call

```
flightData2015.createOrReplaceTempView("flight_data_2015")
```

- Now we can query our data in SQL
- To execute a SQL query, we'll use the `spark.sql()` function that returns a new DataFrame
- While this may seem a bit circular in logic that a SQL query against a DataFrame returns another DataFrame, it's actually quite powerful
- As a user, you can specify transformations in the manner most convenient to you at any given point in time and not have to trade any efficiency to do so



# DataFrames and SQL

```
sqlWay = spark.sql("""  
SELECT DEST_COUNTRY_NAME, count(1)  
FROM flight_data_2015  
GROUP BY DEST_COUNTRY_NAME  
""")
```

```
dataFrameWay = flightData2015\  
.groupBy("DEST_COUNTRY_NAME")\  
.count()
```

# DataFrames and SQL

```
maxSql = spark.sql("""
SELECT DEST_COUNTRY_NAME, sum(count) as destination_total
FROM flight_data_2015
GROUP BY DEST_COUNTRY_NAME
ORDER BY sum(count) DESC
LIMIT 5
""")
maxSql.collect()
```

```
from pyspark.sql.functions import desc
flightData2015\
.groupBy("DEST_COUNTRY_NAME")\
.sum("count")\
.withColumnRenamed("sum(count)", "destination_total")\
.sort(desc("destination_total"))\
.limit(5)\
.collect()
```

# Using Spark SQL (DataFrames)

- Python API docs
  - <https://spark.apache.org/docs/2.4.3/api/python/index.html>
  - Use 'Quick Search' to find the operation for which you are looking
- Or see “Spark – Python API – SQL & DataFrames” on the blackboard in the “Free Books and Chapters” section

# Using Spark SQL (DataFrames)

- `pyspark.sql.Session`
  - Main entry point for DataFrame and SQL functionality
  - A `Session` can be used to...
    - Create DataFrame
    - Execute SQL over tables
    - Cache tables

# Using Spark SQL (DataFrames)

- `pyspark.sql.DataFrameReader`
  - Interface used to load a DataFrame from external storage systems  
Use `SparkSession.read()` to access this
- `pyspark.sql.DataFrameWriter`
  - Interface used to write a DataFrame to external storage system
  - Use `DataFrame.write()` to access this.

# Using Spark SQL (DataFrames)

- `pyspark.sql.DataFrame`
  - A distributed collection of data grouped into named columns
  - A DataFrame is equivalent to a relational table in Spark SQL
  - Can register a DataFrame as table
- `pyspark.sql.Row`
  - A row of data in a DataFrame
- `pyspark.sql.StructType`
  - Defines the schema of (each Row in) a DataFrame
- `pyspark.sql.StructField`
  - Defines the name and type of a field in (each Row in) a DataFrame
- `pyspark.sql.types`
  - List of data types available for each StructField

# SparkContext and SparkSession

- SparkContext was the main entry point for an application using the Spark Core API (that is work with RDDs)
- In pyspark an instance of SparkContext is available and called 'sc'
- SparkSession is the main entry point for Spark SQL (and Spark DataFrames)
- In pyspark an instance of SparkSession is available and called 'spark'

# DataFrame API

- In a previous lecture, you learned how to manipulate RDDs
- This is important because RDDs represent a low-level, direct way of manipulating data in Spark and the core of Spark runtime
- Spark 1.3 introduced the DataFrame API for handling structured, distributed data in a table-like representation with named columns and declared column types



# DataFrame API

- A DataFrame is an RDD that has a schema
- You can think of it as a relational database table, in that each column has a name and a known type
- The power of DataFrames comes from the fact that, when you create a DataFrame from a structured dataset...
- Spark is able to infer a schema by making a pass over the entire dataset that's being loaded
- When calculating the execution plan, Spark can use the schema and do better computation optimizations
- Alternatively you can programmatically associate a schema with a DataFrame

# DataFrame API

- DataFrames can be constructed from a wide range of sources including...
  - An existing RDD
  - A JSON file
  - A text file, Parquet file, or ORC file
  - A table in Hive
  - An external database

# DataFrame API

- As an example, the following creates a DataFrame based on the content of a JSON file:

```
df = spark.read.json("hdfs://examples/src/main/resources/people.json")
```

```
# Displays the content of the DataFrame to stdout
```

```
df.show()
```

# DataFrame API

- DataFrames translate SQL code and domain-specific language (DSL) expressions into optimized low-level RDD operations...
- So that the same API can be used from any supported language (Scala, Java, Python, and R) for accessing any supported data source (files, databases, and so forth) in the same way and with comparable performance characteristics
- Since their introduction, DataFrames have become one of the most important features in Spark and made Spark SQL the most actively developed Spark component
- Since Spark 2.0, DataFrame is implemented as a special case of DataSet.

# DataFrame API

- SQL is so ubiquitous that the DataFrame API quickly met with acclamation by the wider Spark community
- It allows you to attack a problem from a higher vantage point when compared to Spark Core transformations
- The SQL-like syntax lets you express your intent in a more declarative fashion...
- You describe what you want to achieve with a dataset, whereas with the Spark Core API you basically specify how to transform the data (to reshape it so you can come to a useful conclusion)
- You may therefore think of Spark Core as a set of fundamental building blocks on which all other facilities are built
- The code you write using the DataFrame API gets translated to a series of Spark Core transformations under the hood.

# Schemas

- A schema defines the column names and types of a DataFrame
- We can either let a data source define the schema or we can define it explicitly ourselves
- The function `schema` returns the schema associated with a DataFrame (whether Spark inferred or we provided the schema definition)

```
spark.read.format("json")\  
.load("hdfs:///user/hadoop/2015-summary.json")\  
.schema(...)
```

# Schemas


Python will return

Column Name

Column Type

Null Allowed?

```
StructType(  
    List (  
        StructField(DEST_COUNTRY_NAME, StringType, true),  
        StructField(ORIGIN_COUNTRY_NAME, StringType, true),  
        StructField(count, LongType, true)  
    )  
)
```



- A schema is a StructType made up of a number of fields called StructFields
- That provide a name, type, and a Boolean flag which specifies whether or not that column can contain missing or null values

# Schemas

- Here's how to create, and enforce a specific schema on a DataFrame
- If the types in the data (on read), do not match the schema then Spark will throw an error

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

```
myManualSchema = StructType([  
    StructField("DEST_COUNTRY_NAME", StringType(), True),  
    StructField("ORIGIN_COUNTRY_NAME", StringType(), True),  
    StructField("count", LongType(), False)  
])
```

```
df = spark.read.format("json")\  
    .schema(myManualSchema)\  
    .load("/hdfs:///user/hadoop/2015-summary.json")
```



# Programmatically Specifying the Schema

- The preferred method of defining a schema for your DataFrame objects is to explicitly supply this in your code
- Creating a schema requires you to create a StructType object containing a collection of StructField object
- Apply this schema to your DataFrame when it is created

# Programmatically Specifying the Schema

- `class pyspark.sql.types.StructType(fields=None)`
  - Struct type, consisting of a list of `StructField`.
  - This is the data type representing a Row.

# Programmatically Specifying the Schema

- `class pyspark.sql.types.StructField(name, dataType, nullable=True)`
  - A field in StructType.
  - Parameters:
    - `name` – string, name of the field.
    - `dataType` – `DataType` of the field.
    - `nullable` – boolean, whether the field can be null (`None`) or not.

# Programmatically Specifying the Schema

```
# Construct the schema
```

```
fields = [StructField('name', StringType(), True),  
          StructField('age', StringType(), True)]  
schema = StructType(fields)
```

```
# Apply the schema to the RDD.
```

```
schemaPeople = spark.createDataFrame(people, schema)
```

```
# Register the DataFrame as a table.
```

```
schemaPeople.createOrReplaceTempView("people")
```

```
# SQL can be run over DataFrames that have been registered
```

```
# as a table.
```

```
results = spark.sql("SELECT name FROM people")
```

# DataFrame Type Model

- The data model for the DataFrame API is based on the Hive data model
- Datatypes used with DataFrames map directly to their equivalents in Hive
- This includes all of the common primitive types as well as the complex types

# DataFrame Primitive Types

Type	Hive Equivalent	Python Equivalent
ByteType	TINYINT	int
ShortType	SMALLINT	int
IntegerType	INT	int
LongType	BIGINT	long
FloatType	FLOAT	float
DoubleType	DOUBLE	float
BooleanType	BOOLEAN	bool
StringType	STRING	string
BinaryType	BINARY	bytearray
TimestampType	TIMESTAMP	datetime.datetime
DateType	DATE	datetime.date

# DataFrame Complex Types

Type	Hive Equivalent	Python Equivalent
ArrayType	ARRAY	list, tuple, or array
MapType	MAP	dict
StructType	STRUCT	list or tuple

# Running SQL Queries on DataFrames

- The `sql()` function of `SparkSession` enables applications to run SQL queries and returns the result as a `DataFrame`
- SQL queries reference tables and not `DataFrames`...
- So `DataFrame` provides a function to register itself as a table known to the SQL processor



# DataFrame

## `createOrReplaceTempView(tableName)`

- Register as a temporary table in the catalog.
- The lifetime of the table is tied to that of the SparkSession used to create the associated DataFrame

```
someDataFrame.createOrReplaceTempView( "table1")
```

# SparkSession

## sql(sqlQuery)

- Returns a DataFrame representing the result of the given query
- Returns:     DataFrame

```
df1.createOrReplaceTempView("table1")
```

```
df2 = spark.sql("SELECT field1 AS f1, field2 as f2 from table1")
```

```
df2.collect()
```

```
[Row(f1=1, f2=u'row1'), Row(f1=2, f2=u'row2'), Row(f1=3, f2=u'row3')]
```

# Creating DataFrames

- There are six core data sources from which you can create a DataFrame
  - CSV
  - JSON
  - Parquet
  - ORC
  - JDBC/ODBC
  - Plain text files
- For testing purposes you can even create a DataFrame from a list programmatically
- Spark also has many community developed data sources from which a DataFrame can be created
  - Cassandra
  - MongoDB
  - Many more

# Creating DataFrames

## From a Python List

The simplest way to create a DataFrame is from a Python list of data:  
`SparkSession.createDataFrame(data, schema=None)`

```
simpleData = [  
    ("James",34,"2006-01-01","true","M",3000.60),  
    ("Michael",33,"1980-01-10","true","F",3300.80),  
    ("Robert",37,"06-01-1992","false","M",5000.50)  
]  
  
columns = ["firstname","age","jobStartDate","isGraduated","gender","salary"]  
df = spark.createDataFrame(data = simpleData, schema = columns)
```

# Creating DataFrames

## From a Python List

The simplest way to create a DataFrame is from a Python list of data:  
`SparkSession.createDataFrame(data, schema=None)`

```
data2 = [("James", "", "Smith", "36636", "M", 3000),  
        ("Michael", "Rose", "", "40288", "M", 4000),  
        ("Robert", "", "Williams", "42114", "M", 4000)  
]
```

```
schema = StructType([\br/>    StructField("firstname", StringType(), True), \br/>    StructField("middlename", StringType(), True), \br/>    StructField("lastname", StringType(), True), \br/>    StructField("id", StringType(), True), \br/>    StructField("gender", StringType(), True), \br/>    StructField("salary", IntegerType(), True) \br/>])
```

```
df = spark.createDataFrame(data=data2, schema=schema)
```

# Creating DataFrames

## The core structure for reading external data

- The foundation for reading external data into Spark is the `DataFrameReader`
- We access a `DataFrameReader` from the `SparkSession` via the `read` attribute  
`spark.read`
- After we have a `DataFrame` reader, we specify several values:
  - The format (of the data to be read in, default is `parquet`)
  - The schema (if any, by which the data is described, default is `none` and Spark infers a schema)
  - The read mode (indicating how to handle malformed data, default is `permissive`)
  - A series of options
- The format, options, and schema each return a `DataFrameReader` that can undergo further transformations
- The general pattern for reading external data looks as follows  
`spark.format(...).option("key1", "value1").option("key2", "value2").schema(...).load(...)`

# Creating DataFrames

## The core structure for reading external data

- As an example for reading in a csv file

```
spark. \  
    read.format("csv") \  
    .option("mode", "FAILFAST") \  
    .option("header", True) \  
    .schema(someSchema) \  
    .load("hdfs://user/hadoop/somefile.csv")
```

- But there are shorthand way that are generally used to read in data having common formats, which we will use going forward, for example:

```
someSchema = ...  
spark.read.csv("hdfs:///path/to/file", schema=someSchema, sep=':')
```

# Creating DataFrames from Files

- Version 1.4 of Spark introduced a new interface used to load DataFrames from external storage systems
- This interface is provided through the DataFrameReader class
- The DataFrameReader interface is accessed using the `SparkSession.read()` function



# SparkSession

## read()

- Returns a DataFrameReader that can be used to read data in as a DataFrame.
- Returns:     DataFrameReader

# DataFrameReader

## json(path, schema=None)

- Loads a JSON file (one object per line) or an RDD of Strings storing JSON objects (one object per record) and returns the result as a `:class`DataFrame``
- JSON is a common, standard, human-readable serialization (or wire transfer) format that is often used in web service responses
- Parameters:
  - path – string represents path to the JSON dataset, or RDD of Strings storing JSON objects.
  - schema – an optional StructType for the input schema
  - If the schema parameter is not specified, this function goes through the input once to determine the input schema.
- Example

```
df1 = spark.read.json('hdfs:///user/CSP554prof/people.json')
```

# DataFrameReader

## text(paths)

- Loads a text file and returns a DataFrame with a single string column named “value”
- Each line in the text file is a new row in the resulting DataFrame.

- Parameters:

- paths – string, or list of strings, for input path(s).

- Example

```
df = spark.read.text('python/test_support/sql/text-test.txt')
```

```
df.collect()
```

```
[Row(value=u'hello'), Row(value=u'this')]
```

# DataFrameReader

## Other File Types

- `parquet(*paths)`
  - Loads a Parquet file, returning the result as a DataFrame
- `orc(path)`
  - Loads an ORC file, returning the result as a DataFrame
  - Note: Currently ORC support is only available together with HiveContext
- `jdbc(...)`
  - Construct a DataFrame representing the database table accessible via JDBC URL *url* named *table* and connection *properties*

# Creating a DataFrame from a Hive Table

- To load data from a Hive table into a Spark SQL DataFrame, you will need a HiveContext to be created
- HiveContext reads the Hive client configuration (hive-site.xml) to obtain connection details for the Hive metastore
- This enables seamless access to Hive tables from a Spark application

# Creating a DataFrame from a Hive Table

- When working with Hive one must construct a HiveContext, which inherits from SparkSession, and adds support for finding tables in the MetaStore and writing queries using HQL.

```
# sc is an existing SparkContext.
```

```
from pyspark.sql import HiveContext
```

```
SparkSession = HiveContext(sc)
```

```
spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")
```

```
spark.sql("LOAD DATA LOCAL INPATH 'user/cs/kv1.txt' INTO TABLE src")
```

```
# Queries can be expressed in HiveQL.
```

```
results = spark.sql("FROM src SELECT key, value").collect()
```

# DataFrame Operations/Transformations

- DataFrames provide a domain-specific language for data manipulation in Scala, Java, Python and R
- Here we include some basic examples of structured data processing using DataFrames...

# Transformations

- In Spark, the core data structures are immutable meaning they cannot be changed once created.
- If you cannot change it, how are you supposed to use it?
- In order to “change” a DataFrame you will have to instruct Spark how you would like to modify the DataFrame you have (the “source”) into the one that you want (the “target”)
- These instructions are called transformations.
- Let’s perform a simple transformation to find all even numbers in our current DataFrame

```
targetDF = sourceDF.where("number % 2 = 0")
```

- Note, this code returns no output, that’s because we only specified an abstract transformation and Spark will not act on transformations until we call an action



# Transformation Types

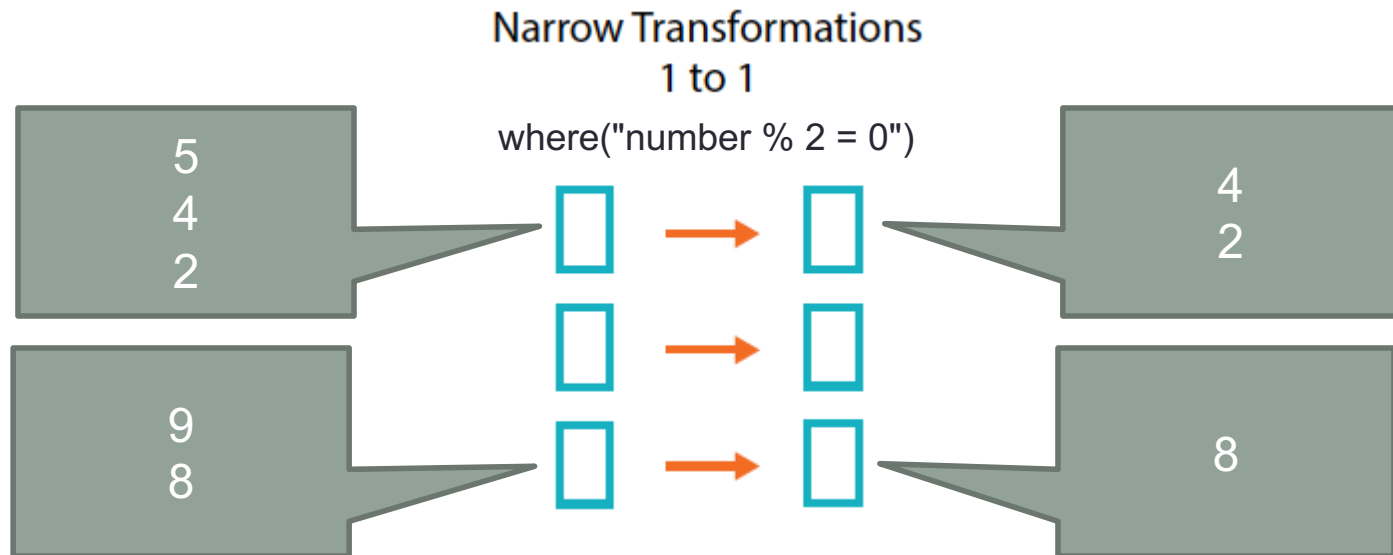
- There are two types of transformations...
- Those that specify narrow dependencies
- And those that specify wide dependencies

# Narrow Transformations

- Transformations consisting of narrow dependencies (we'll call them narrow transformations) are those where each input partition will contribute to only one output partition
- In the preceding code snippet our “where” statement specifies a narrow dependency

```
targetDF = sourceDF.where("number % 2 = 0")
```

- Only one input partition contributes to at most one output partition

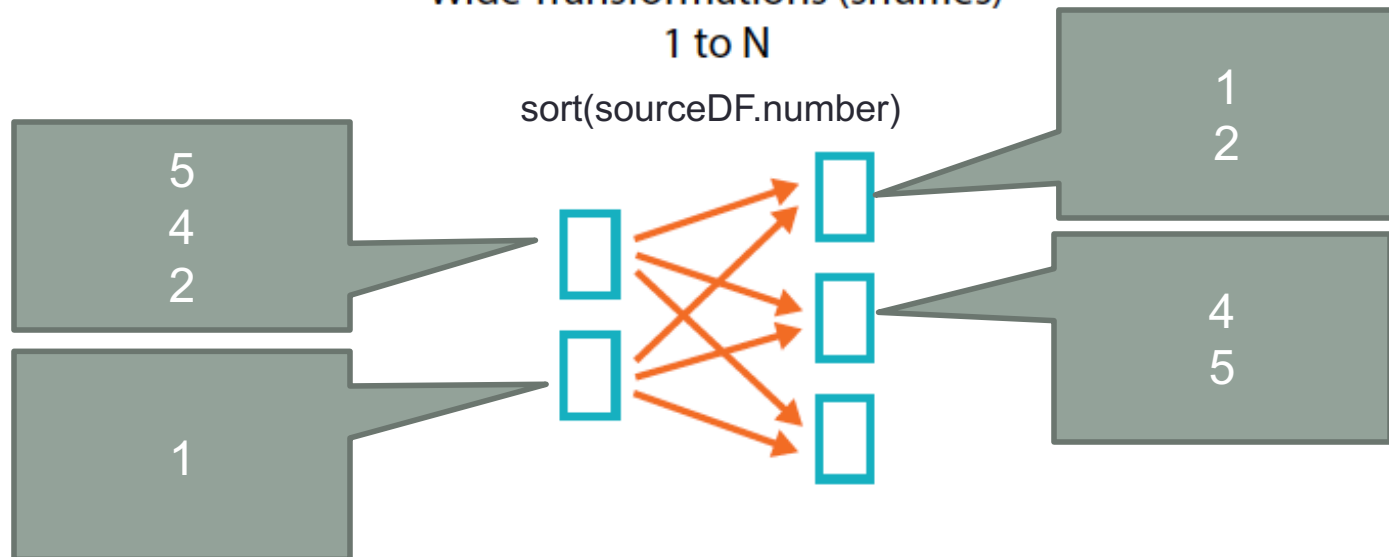


# Wide Transformations

- A wide dependency (or wide transformation) style transformation will have input partitions contributing to many output partitions
  - Examples include the `join()`, `sort()` and `groupByKey()` transformations
- You will often hear this referred to as a shuffle where Spark will exchange partitions across the cluster
- When we perform a shuffle. Spark will write the results to disk

`targetDF = sourceDF.sort(sourceDF.number)`

Wide Transformations (shuffles)



# Lazy Evaluation

- Both the above Narrow Wide Transformation types are lazy in nature...
- This means that until and unless any action is performed over these transformations their execution is delayed
- Due to this delay the Spark execution engine gets a whole view of all the chained transformations and ample time to optimize your query
- Spark, by waiting until the last minute to execute the code, will compile your raw, DataFrame transformations, to an efficient physical plan that will run as efficiently as possible across the cluster
- This provides immense benefits to the end user because Spark can optimize the entire data flow from end to end.

# Actions

- Transformations allow us to build up our logical transformation plan
- To trigger the computation, we run an action
- An action instructs Spark to compute a result from a series of preceding transformations
- The simplest action is count which gives us the total number of records in the DataFrame

```
recordCount = sourceDF.count()
```

*Note recordCount is a simple numeric value, and not a DataFrame*

- There are three kinds of actions:
  - actions to view data in the console
  - actions to collect data to native objects in the respective language
  - and actions to write to output data sources.

# Examples

- Transformations contribute to the query plan, but they don't execute anything
- Actions cause the execution of the query

## **Transformation examples**

- filter
- select
- drop
- intersect
- join

## **Action examples**

- count
- collect
- show
- head
- take

# What can I do with a DataFrame?

- Once you have a DataFrame, there are a number of operations you can perform
- Let's look at a few of them
- But, first, let's talk about columns
- When we say “column” here, what do we mean?
- A DataFrame column is an abstraction
- It provides a common column-oriented view of the underlying data
  - Regardless of how the data is really organized

# Columns

Let's see how  
DataFrame  
columns map  
onto some  
common data  
sources

Input Source Format	Data Frame Variable Name	Data									
JSON	<b>dataFrame1</b>	[ {"first": "Amy", "last": "Bello", "age": 29 }, {"first": "Ravi", "last": "Agarwal", "age": 33 }, ... ]									
CSV	<b>dataFrame2</b>	first,last,age Fred,Hoover,91 Joaquin,Hernandez,24 ...									
SQL Table	<b>dataFrame3</b>	<table> <tr> <th>first</th><th>last</th><th>age</th></tr> <tr> <td>Joe</td><td>Smith</td><td>42</td></tr> <tr> <td>Jill</td><td>Jones</td><td>33</td></tr> </table>	first	last	age	Joe	Smith	42	Jill	Jones	33
first	last	age									
Joe	Smith	42									
Jill	Jones	33									



# Columns

Input Source Format	Data Frame Variable Name	Data									
JSON	<b>dataFrame1</b>	<pre>[ {"first": "Amy",   "last": "Bello",   "age": 29 },   {"first": "Ravi",   "last": "Agarwal",   "age": 33 },   ... ]</pre>									
CSV	<b>dataFrame2</b>	<pre>first,last,age Fred,Hoover,91 Joaquin,Hernandez,24 ...</pre>									
SQL Table	<b>dataFrame3</b>	<table> <thead> <tr> <th>first</th><th>last</th><th>age</th></tr> </thead> <tbody> <tr> <td>Joe</td><td>Smith</td><td>42</td></tr> <tr> <td>Jill</td><td>Jones</td><td>33</td></tr> </tbody> </table>	first	last	age	Joe	Smith	42	Jill	Jones	33
first	last	age									
Joe	Smith	42									
Jill	Jones	33									

**dataFrame1**  
column: "first"

**dataFrame2**  
column: "first"

**dataFrame3**  
column: "first"

# Columns

- Assume we have a DataFrame, `df`, that reads a data source that has "first", "last", and "age" columns

Python	Java	Scala	R
<code>df["first"]</code> <code>df.first<sup>†</sup></code>	<code>df.col("first")</code>	<code>df("first")</code> <code>\$"first"<sup>‡</sup></code>	<code>df\$first</code>

- In Python, it's possible to access a DataFrame's columns either by attribute (`df.age`) or by indexing (`df['age']`)
- While the former is convenient for interactive data exploration, you should use the index form
- It won't break with column names that are also attributes on the DataFrame class.

# DataFrame Operations

## Examples

- Assume we have a file of age and name information (person.json) encoded in JSON format
- Note, for Spark SQL a JSON record must occur on one line

```
{"name":"Michael"}
```

```
{"name":"Andy", "age":30}
```

```
{"name":"Justin", "age":19}
```

# DataFrame Operations

## Examples: Create a DataFrame from a File

```
from pyspark.sql import SparkSession  
spark = SparkSession(sc)
```

```
# Create the DataFrame
```

```
df = spark.read.json("hdfs:///user/CSP554prof/people.json")
```

- The read function of the SparkSession creates a DataFrameReader whose json() function loads the indicated file into a DataFrame
- On loading the DataFrame schema is inferred from the input file

# DataFrame Operations

## Examples: Show the Content of the DataFrame

- You can look at the first  $n$  elements in a DataFrame with the `show(n=20)` method
  - If not specified,  $n$  defaults to 20
  - **This method is an action**
- `show()`...
  - Reads the input source
  - Executes the RDD DAG across the cluster
  - Pulls the  $n$  elements back to the driver JVM
  - Displays those elements in a tabular form

# DataFrame Operations

Examples: Show the Content of the DataFrame

`df.show()`

- Output

age name

null Michael

30 Andy

19 Justin

# DataFrame Operations

## Examples: Print the Schema in a Tree Format

```
df.printSchema()
```

You can have Spark tell you what it thinks the data schema is, by calling the `printSchema()` method

- Output

```
root
```

```
|-- age: long (nullable = true)
```

```
|-- name: string (nullable = true)
```

# DataFrame Operations

## collect()

- The collect() action is used to retrieve all the elements of a DataFrame to the driver node
- Use collect() on smaller dataset usually after filter(), group(), count() etc.
- Retrieving larger dataset results in out of memory



# DataFrame Operations

## collect()

```
dept = [("Finance",10), \
        ("Marketing",20), \
        ("Sales",30), \
        ("IT",40) \
        ]
```

```
deptColumns = ["dept_name","dept_id"]
```

```
deptDF = spark.createDataFrame(data=dept, schema = deptColumns)
```

```
deptDF.show(truncate=False)
```

```
+-----+-----+
|dept_name|dept_id|
+-----+-----+
|Finance  |10     |
|Marketing|20     |
|Sales    |30     |
|IT       |40     |
+-----+-----+
```

# DataFrame Operations

## collect()

```
dataCollect = deptDF.collect()  
print(dataCollect)
```

deptDF.collect() retrieves all elements in a DataFrame as an array or Rows to the driver

Printing a resultant array yields below output:

```
[Row(dept_name='Finance', dept_id=10),  
Row(dept_name='Marketing', dept_id=20),  
Row(dept_name='Sales', dept_id=30),  
Row(dept_name='IT', dept_id=40)]
```

# DataFrame Operations

## select()

- `select()` is like a SQL `SELECT`, allowing you to limit the results to specific columns

```
df.select(df['first_name'], df['age'], df['age'] > 49).show(5)
```

```
+-----+-----+-----+
|first_name|age|(age > 49)|
+-----+-----+-----+
|      Erin| 42|      false|
|    Claire| 23|      false|
|    Norman| 81|       true|
|    Miguel| 64|       true|
|Rosalita  | 14|      false|
+-----+-----+-----+
```

# DataFrame

## `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.

```
>>> df.select('*').collect()
[Row(age=2, name=u'Alice'), Row(age=5, name=u'Bob')]
```

```
>>> df.select('name', 'age').collect()
[Row(name=u'Alice', age=2), Row(name=u'Bob', age=5)]
```

```
>>> df.select(df.name, (df.age + 10).alias('age')).collect()
[Row(name=u'Alice', age=12), Row(name=u'Bob', age=15)]
```

# DataFrame Operations

## Examples: Select Only the "name" Column

```
df.select('name').show()
```

or

```
df.select(df.name).show()
```

or

```
df.select(df['name']).show()
```

- Output

name

Michael

Andy

Justin

# DataFrame Operations

## filter() or where()

- The filter() method allows you to filter rows out of your results
- where() is an alias for filter()

```
df.filter(df['age'] > 49).\n    select(df['first_name'], df['age']).show()
```

```
+-----+---+
|firstName|age|
+-----+---+
|   Norman| 81|
|   Miguel| 64|
|  Abigail| 75|
+-----+---+
```

# DataFrame Operations

## Examples: Select People Older Than 21

```
df.filter(df['age'] > 21).show()
```

- Output  
age name  
30 Andy

# DataFrame Operations

## orderBy()

- The orderBy() method allows you to sort the results

```
df.filter(df["age"] > 49). \
  select(df["firstName"], df["age"]). \
  orderBy(df["age"], df["firstName"]). \
  show()
```

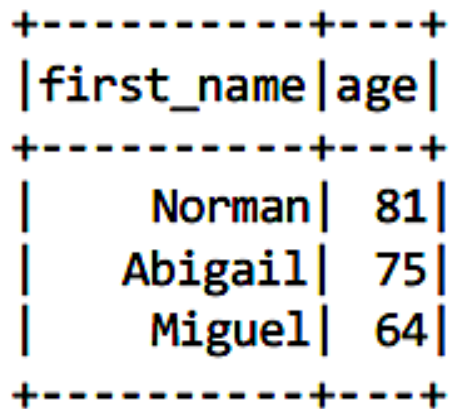
```
+-----+-----+
|firstName|age|
+-----+-----+
|   Miguel| 64|
|  Abigail| 75|
|   Norman| 81|
+-----+-----+
```



# DataFrame Operations

## orderBy()

```
df.filter(df['age'] > 49).\n  select(df['first_name'], df['age']).\n  orderBy(df['age'].desc(), df['first_name']).show()
```



first_name	age
Norman	81
Abigail	75
Miguel	64

- `desc()`
  - Returns a sort expression based on the descending order of the column
- `asc()`
  - (default) Returns a sort expression based on ascending order of the column

# DataFrame Operations

## withColumn()

Returns a new DataFrame by adding a column or replacing the existing column that has the same name

`withColumn(colName, col)`

- **colName** – string, name of the new column
- **col** – a column expression for the new column

```
df.withColumn('age2', df['age'] + 2)
```

# DataFrame Operations

## withColumnRenamed()

- Returns a new DataFrame by renaming an existing column
- `withColumnRenamed(existing, new)`
- `existing` – string, name of the existing column to rename
- `new` – string, new name of the column

```
df.withColumnRenamed('age', 'age2')
```

# DataFrame Operations

## cast()

- cast(dataType)
- Convert the column into type datatype

```
df.select(df.age.cast(StringType()).alias('ages'))
```

# DataFrame Operations

## alias()

- alias() allows you to rename a column
- It's especially useful with generated columns

```
df.select(df['first_name'], \
          df['age'], \
          (df['age'] < 30).alias('young')).show(5)
```

```
+-----+-----+
|first_name|age|young|
+-----+-----+
|      Erin| 42|false|
|    Claire| 23| true|
|    Norman| 81|false|
|    Miguel| 64|false|
|  Rosalita| 14| true|
+-----+-----+
```

# DataFrame Operations

## groupBy()

- groupBy(cols)
- Groups the DataFrame using the specified columns, so we can run aggregations, like count(), on them
- cols – list of columns to group by. Each element should be a column name (string) or a column expression
- groupBy() returns an instance of GroupData which offers a range of aggregation functions including count()

```
df.groupBy("age").count().show()
```

```
+---+-----+
|age|count|
+---+-----+
| 39|    1|
| 42|    2|
| 64|    1|
| 75|    1|
| 81|    1|
| 14|    1|
| 23|    2|
+---+-----+
```

# DataFrame Operations

## groupBy()

- This example does a groupBy() on department column and calculates sum() and avg() of salary for each department and calculates sum() and max() of bonus for each department

```
df.groupBy("department") \
    .agg(sum("salary").alias("sum_salary"), \
    avg("salary").alias("avg_salary"), \
    sum("bonus").alias("sum_bonus"), \
    max("bonus").alias("max_bonus") \
    ) \
.show(truncate=False)
```

department	sum_salary	avg_salary	sum_bonus	max_bonus
Sales	257000	85666.66666666667	53000	23000
Finance	351000	87750.0	81000	24000
Marketing	171000	85500.0	39000	21000

# Aggregation Functions

```
df.groupBy(...).someAggregationFunction(cols)
```

When we perform `groupBy()` on a Dataframe, it returns a `GroupData` instance which contains below aggregate functions:

- `count(*cols)` - Returns the count of rows for each group.
- `mean(*cols)` - Returns the mean of values for each group.
- `max(*cols)` - Returns the maximum of values for each group.
- `min(*cols)` - Returns the minimum of values for each group.
- `sum(*cols)` - Returns the total for values for each group.
- `avg(*cols)` - Returns the average for values for each group.
- `agg(*cols)` - Using `agg()` function, we can calculate more than one aggregate at a time.



# DataFrame

## describe(\*cols)

- Computes statistics for numeric columns.
- This include count, mean, stddev, min, and max
- If no columns are given, this function computes statistics for all numerical columns

# DataFrame

## describe(\*cols)

```

• >>> df.describe().show()
• +-----+-----+
• |summary|      age|
• +-----+-----+
• | count  |      2|
• | mean   |     3.5|
• | stddev | 2.1213203435596424|
• | min    |      2|
• | max    |      5|
• +-----+-----+
• >>> df.describe(['age', 'name']).show()
• +-----+-----+-----+
• |summary|    age| name|
• +-----+-----+-----+
• | count|      2|  2|
• | mean|     3.5| null|
• | stddev| 2.1213203435596424| null|
• | min|      2| Alice|
• | max|      5|  Bob|
• +-----+-----+-----+

```

# DataFrame

## distinct()

- Returns a new DataFrame containing the distinct rows in this DataFrame

```
df.distinct().count()
```

```
2
```

# DataFrame

## drop(col)

- Returns a new DataFrame that drops the specified column
- Parameters:
  - col – a string name of the column to drop, or a Column to drop.

# DataFrame

## drop(\*cols)

```
>>> df.drop('age').collect()
```

```
[Row(name=u'Alice'), Row(name=u'Bob')]
```

```
>>> df.drop(df.age).collect()
```

```
[Row(name=u'Alice'), Row(name=u'Bob')]
```

```
>>> df.join(df2, df.name == df2.name, 'inner').drop(df.name).collect()
```

```
[Row(age=5, height=85, name=u'Bob')]
```

```
>>> df.join(df2, df.name == df2.name, 'inner').drop(df2.name).collect()
```

```
[Row(age=5, name=u'Bob', height=85)]
```

# DataFrame

## dropna(how='any', thresh=None, subset=None)

- Returns a new DataFrame omitting rows with null values
- Parameters:
  - how – 'any' or 'all'. If 'any', drop a row if it contains any nulls. If 'all', drop a row only if all its values are null.
  - thresh – int, default None If specified, drop rows that have less than thresh non-null values. This overwrites the how parameter.
  - subset – optional list of column names to consider.
- >>> df4.dropna().show()
- +---+-----+-----+
- |age|height| name|
- +---+-----+-----+
- | 10| 80|Alice|
- +---+-----+-----+

# DataFrame

## dtypes

- dtypes
- Returns all column names and their data types as a list

```
>>> df.dtypes
```

```
[('age', 'int'), ('name', 'string')]
```

# DataFrame

## fillna(value, subset=None)

- Replace null values
- Parameters:
  - value – int, long, float, string, or dict. Value to replace null values with. If the value is a dict, then subset is ignored and value must be a mapping from column name (string) to replacement value. The replacement value must be an int, long, float, or string.
  - subset – optional list of column names to consider
    - Columns specified in subset that do not have matching data type are ignored
    - For example, if value is a string, and subset contains a non-string column, then the non-string column is simply ignored.



# DataFrame

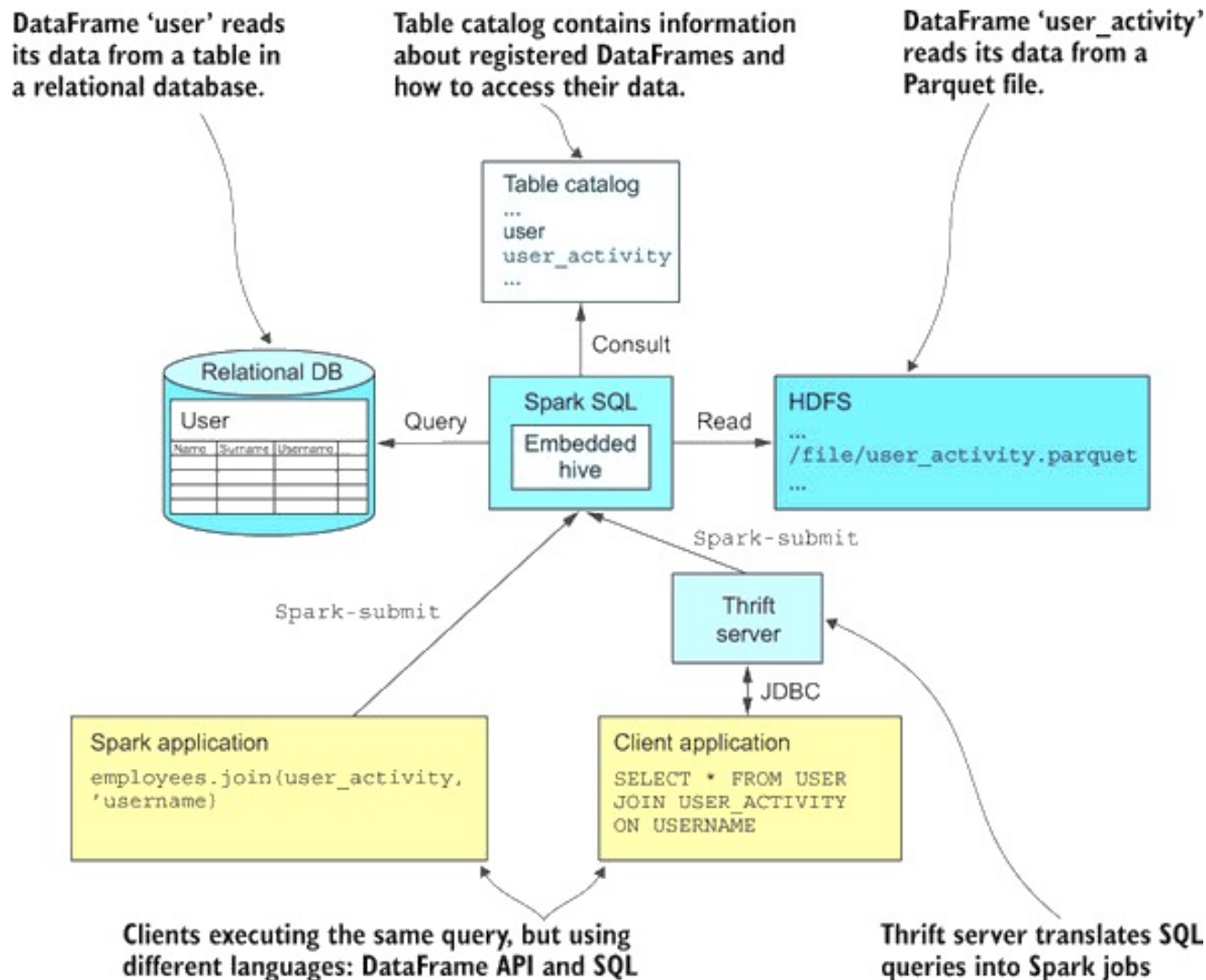
## filter(condition)

- Filters rows using the given condition.
- Parameters:
  - condition – a Column of types.BooleanType or a string of SQL expression

# DataFrame

## filter(condition)

- `>> df.filter(df.age > 3).collect()`
- `[Row(age=5, name=u'Bob')]`
  
- `>>> df.where(df.age == 2).collect()`
- `[Row(age=2, name=u'Alice')]`
  
- `>>> df.filter("age > 3").collect()`
- `[Row(age=5, name=u'Bob')]`
  
- `>>> df.where("age = 2").collect()`
- `[Row(age=2, name=u'Alice')]`



# DataFrame

## first()

- Returns the first row as a Row.
- `>>> df.first()`
- `Row(age=2, name=u'Alice')`

# DastaFrame

## foreach()

- Applies the f function to all Row of this DataFrame.
- ```
>>> def f(person):
```
- ```
...     print(person.name)
```
- ```
>>> df.foreach(f)
```

# DataFrame

## head(n=None)

- Returns the first n rows.
- Parameters
  - n – int, default 1. Number of rows to return.
- Returns
  - If n is greater than 1, return a list of Row. If n is 1, return a single Row.

```
>>> df.head()
Row(age=2, name=u'Alice')
>>> df.head(1)
[Row(age=2, name=u'Alice')]
```

# Other Useful Transformations

| Method                                   | Description                                                                                                                                                              |
|------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>limit(<i>n</i>)</code>             | Limit the results to <i>n</i> rows. <code>limit()</code> is not an action, like <code>show()</code> or the RDD <code>take()</code> method. It returns another DataFrame. |
| <code>distinct()</code>                  | Returns a new DataFrame containing only the unique rows from the current DataFrame                                                                                       |
| <code>drop(<i>column</i>)</code>         | Returns a new DataFrame with a column dropped. <i>column</i> is a name or a Column object.                                                                               |
| <code>intersect(<i>dataframe</i>)</code> | Intersect one DataFrame with another.                                                                                                                                    |
| <code>join(<i>dataframe</i>)</code>      | Join one DataFrame with another, like a SQL join. We'll discuss this one more in a minute.                                                                               |

# DataFrame

## 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 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 string indicating the name of the join column(s), the column(s) must exist on both sides, and this performs an inner equi-join.
  - how – str, default 'inner'. One of inner, outer, left\_outer, right\_outer, leftsemi



# DataFrame

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

```
.>>> df.join(df2, df.name == df2.name, 'outer').select(df.name,  
df2.height).collect()
```

```
[Row(name=None, height=80), Row(name=u'Alice', height=None),  
Row(name=u'Bob', height=85)]
```

```
>>> cond = [df.name == df3.name, df.age == df3.age]
```

```
>>> df.join(df3, cond, 'outer').select(df.name, df3.age).collect()
```

```
[Row(name=u'Bob', age=5), Row(name=u'Alice', age=2)]
```

```
>>> df.join(df2, 'name').select(df.name, df2.height).collect()
```

```
[Row(name=u'Bob', height=85)]
```

```
>>> df.join(df4, ['name', 'age']).select(df.name, df.age).collect()
```

```
[Row(name=u'Bob', age=5)]
```

# Writing DataFrames

- You can write DataFrames out, as well
- When doing ETL, this is a very common requirement.
- In most cases, if you can read a data format, you can write that data format, as well.
- The core pattern for writing data is as follows:

```
spark.write.format(...).option(...).partitionBy(...).bucketBy(...).sortBy(...).save(...)
```

- But we will generally use more compact forms:
  - `df.write.option("header", True).csv("/tmp/spark_output/zipcodes")`
  - `df.write.format("json").save("/path/to/directory")`

# Writing DataFrames

- The foundation for writing data is the `DataFrameWriter`
- We access the `DataFrameWriter` on a per `DataFrame` basis as follows:

```
someDataFrame.write
```

- Note, you do not specify the name of a file to which to write a `DataFrame` but a directory:  

```
df.write.format("parquet").save("/path/to/directory")
```
- If you're writing to a text file format (e.g., JSON, CSV), you'll typically get multiple output files in that directory
  - Usually one per `DataFrame` partition

# Writing DataFrames

## Write DataFrame to CSV file

- Use the `write()` method of the `DataFrameWriter` object to write a `DataFrame` to a CSV file
- While writing a CSV file you can use several options
  - Use “header” to output the `DataFrame` column names as a header record
  - Use ‘sep’ to specify the delimiter on the CSV output file

```
df.write.option("header",True) \  
.option("sep", '\t') \  
.csv("/tmp/spark_output/zipcodes")
```

# Writing DataFrames

## Saving Modes

- The `DataFrameWriter` also has a method `mode()` to specify saving mode.
  - `overwrite` – mode is used to overwrite the existing file.
  - `append` – To add the data to the existing file.
  - `ignore` – Ignores write operation when the file already exists.
  - `error` or `errorIfExists` – Throw an error if the file already exists. This is the default option.

```
df2.write.mode('overwrite').csv("/tmp/spark_output/zipcodes")
```

//you can also use this

```
df2.write.format("csv").mode('overwrite').save("/tmp/spark_output/zipcodes")
```