**Learning Spark**

Lightning-Fast Data Analytics

CHAPTER 1

Introduction to Apache Spark: A Unified Analytics Engine

This chapter lays out the origins of Apache Spark and its underlying philosophy.

Big Data and Distributed Computing at Google

* Seraching new way to manage Google’s search engine to index and search the world’s data on the internet (traditionnal RDBM’s wasn’t appropriate) : Creation of Google File System + Big table + MapReduce
  + Google File System provided a fault tolerant and distriuted filesystem across many commodity hardware servers in a cluster farm
  + BigTable offered Scalable Storage of structured data across GFS
  + MR introduced a new prallel programming paradigm, base on functionnal programming, for large scale processing of data distributed over GFS and BigTable

The workers in the cluster aggregate and reduce the intermediate computations and produce a final appended output from the reduce function

Sends computation code (map and reduce functions) to where the data resides,

Sends

Output written in distributive storage reachaable by the application

MAP Reduce System

Hadoop at Yahoo!

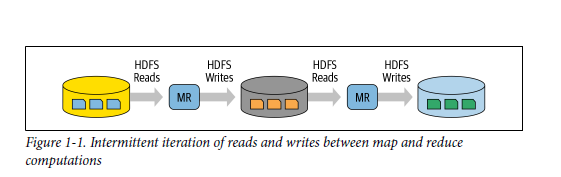
Solution for computation challenges at yahoo was express by GFS and provided the blue print for the Hadoop File System (HDFS) including Map Reduce as frameword for distributed computing => Donated to Apache software foundation in April 2006

Became Apache Hadoop with these modules :

* Hadoop Common,
* MapReduce,
* HDFS,
* Apache Hadoop YARN

Map Reduce has some inconvenient :

* Hard to manage and administer caused of cumbersonne operational complexity
* Generated batch processing, was verbose and lot of boilerplate setup code with britle fault tolerance
* MR operaton was written to the local disk for the subsequent stage and impact disk performance and duration



* Didn’ t work well wth someother workload as machine learning, streaming , SQL like queries (time consuming)

Some alternative, bespoke system (sur mesurre) was prosposed as Apache Hive, Apache Storm, Apache Impala, Apache Giraph, Apache Drill, Apache Mahout, etc.), each with their own APUs and cluster configurations, further adding to the operational complexity of Hafoop

Spark’s Early Years at AMPLab

Research at UC Berkeley took the challenge to male hadoop and MR simpler ad faster called it SPARK in 2009 in RAD Lab

They acknowledged that MR was inefficient (or intractable) for interactive or iterative computing jobs and a complex framework to learn.

Spark today bring the idea borrowed from Hadoop MR but enhance the system:

* Make it highly fault tolerant
* Embarrassingly parallel
* Support in memory storage for intermediate results betwee iteractive and interactive map reduce computations
* Offer easy and composable api in multiple programming languages
* Was open source in may 2014 under apache software fundation

What Is Apache Spark?

**Apache Spark** is a unified engine designed for large scale distributed data processing on premises in data centers or in the cloud

Spark provide :

* In memory storage for intermediate computation (more faster than MR)
* It incorporates libraries with composable API for machine learning (MLlib, SQL for interactive queries(spark SQL), Stream processing (Structured streaming) for interacting with real time data , Graph processing (GraphX)

Spark’s design philosophy center around four key characteristics :

* Speed
* Ease of use
* Modularity
* Extensibility

Speed

Spark gas pursied the goal of speed in several ways :

* Internal implementation (benefits from hardware imprioving in price and performance of CPU and memory)
  + Framework optimized to take advantage of efficient multitreading and parallel processing og unix-based operating system
* Spark builds its query computation as directed acylic graph (DAG), it Scheduler and query optimize construct efficient computational graph, which can be decompose into task executable in parallel across the worker
* Physical execution engine, Tungsten, uses whole stage code generatio to gnenerate compact code for execution
* With Inmemory computation / intermediate results retained in memory, it limit disk I/O give huge performance boost

Ease of Use

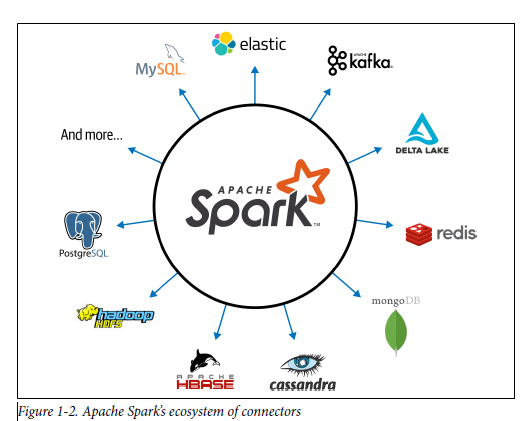
Spark achieves simplicity by providing a fundamental absctraction of a simple logical data strucutre called a Resilient Distributed Dataset (RDD) upon which high levl sutructured data abastractopn, such dataframe, dataset are constructed.  
By provigin set of transformation and actions we got a simple programming model that you can use to build big data application

Modularity

* Spark can be use for many workload
* Supported many programming language(Scala, python, R, SQL, Java)
* Unified labraries, with API with the following modules as core components:
  + Spark sql
  + Spark structured programming
  + Spark MLIB
  + GraphX
* A unified processing engine which run all these librairies

Extensibility

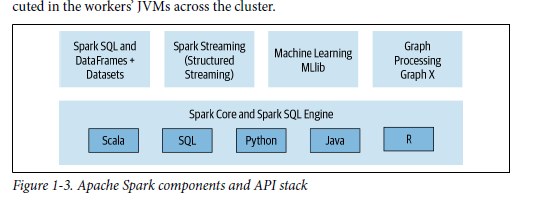
* Spark need to read and write data on a storage, it can sotre on Hadoop, Cassandra, HBase, MongoDB, Hibe, RDBMS…
* Spark community maintains a list of third party spark packages including spark connector for variety of external sources



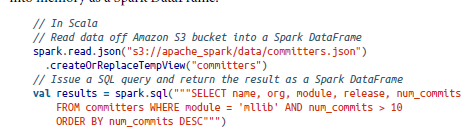
Unified Analytics

“Unified Engine for Big Data Processing.” The award-winning paper notes that Spark replaces all the separate batch processing, graph, stream, and query engines like Storm, Impala, Dremel, Pregel, etc

Apache Spark Components as a Unified Stack



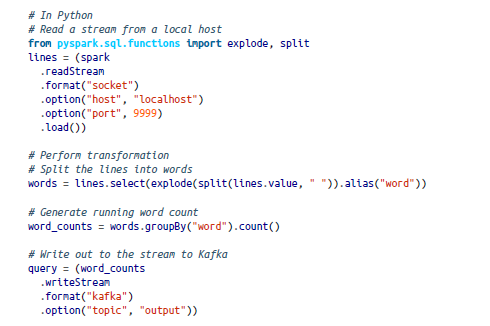
* Spark offers 4 distinct libraires for diverse workloads:
  + Spark SQL:
    - Work well with structured data
    - You can read from RDBMSs, file format with structured data (CSV, text, json, avro, orc, parquet) =>(construct) permanent or temporary tables in spark
    - Spark is ANSI SQL: 2003 compliant
    - Can combine spark structured streaming and spark sql



* + Spark MLlib
    - Spark contains librairies with common machine learning algorithms
    - Machine learning algorithms are built atop highlevel dataframe api
    - API alow
      * Extract or transform feature
      * Build piplines(for training and evaluating)
      * Persit models (for saving and reloading them)
      * MLlib includes other low-level ML primitives, including a generic gradient descent optimization.



* + Spark structured Streaming
    - Apache spark 2 introduce an experimental continuous streaming model and structured streaming APIs
    - Available by Spark 2.2
    - Necessary to combine and react in real time to both static data as streaming data from engines like Apache Kafka
    - View stream as a continually growing table with new row appended at the end
    - Developers can merely treat this as a structured table and issue queries against as it is a static table
    - Spark Sql core engine handles all aspect of fault tolerance and late data semantics (more easier for developer)
    - It replace Dstreams data structure
    - It read from a localhost socket, apache kafka, kinesis and HDFS based/cloud



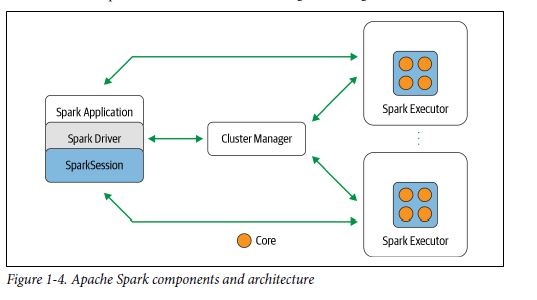
* + GraphX
    - Librairies manipulation graph and perfoming graph parallel computations
    - Standard graph algorithms for analysis, connections and traversals (PageRank, Connected Components, triangle counting..)

Each of these libraries are separate from spark’s core fault tolerant engine :

use APIs(Spark SQL, graphX, MLlib, java ..) =>(spark application convert to) DAG =>(DAG execute by) Core engine

Apache Spark’s Distributed Execution

Spark is a distributed data processing engine with its components working collaboratively on a cluster of machine

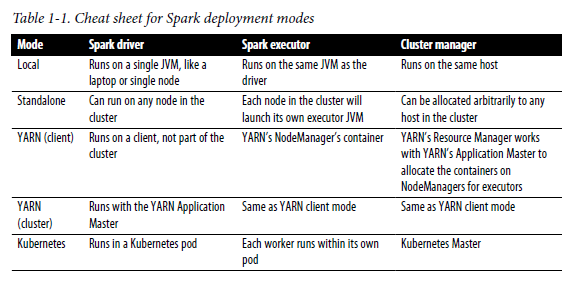


(Art high level) Driver program (orchestrating parallel operatons)=> (access) distributed components in the cluster (spark executors and cluster manager through sparksession)

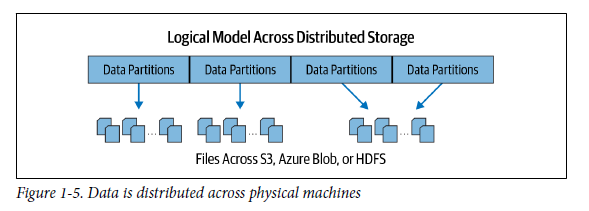
* Spark Driver
  + Instantiating a sparksession ,
  + Communicate with the cluster manager and request resources (CPU, memory etc..)
    - From the cluster manager for spark’s executor (JVM)
  + Transforms all the spark operations into DAG computations, schedules them and distributes their execution as tasks across spark exectuors
  + Once resources are allocated, it commincates directly with the executors
* SparkSession
  + In Spark 2.0 sparksession became a unfied conduit to all spark operations and data (not only subsume previous entry point to spark sparkcontext, sqlcontext, hivecontext, sparkconf and streamingContext but also made working wih spark simpler and easier
  + From this sparksession tou can
    - Create JVM runtime parameter
    - Define dataframes and datasets
    - Read from data sources
    - Access catalog metadata
    - Issue spark sql queries …
  + Can create it using high level api in any languages available
  + It is created autmaticaly when running spark shell and access through spark , sc



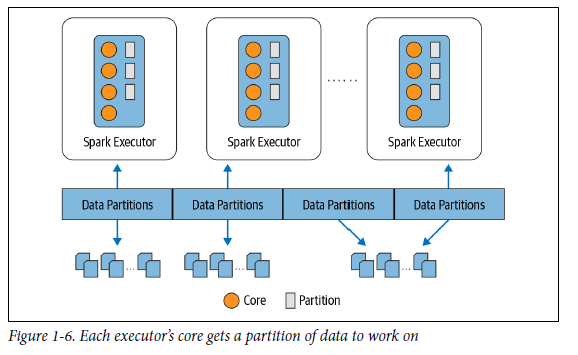
* Cluster manager
  + Responsible for managing and allocating resources for the cluster of node on which your spark application runs (see code sources)
  + Spark support 4 cluster managers:
    - Standalone cluster manager
    - Apache Hadoop YARN
    - Apache Mesos
    - Kubernetes
* Spark Executor
  + A spark executor runs on each worker node in the cluster
  + Communicationg with the dirver program and responsible for executing tasks on the workers in most deplouments modes, only a single executor runs per node
* Deployment modes



* Distrbuted data and partitions
  + Data is distributed ad partitions across the physical cluster
  + Spark treats each partition as a high level logical data abstraction as DataFrame in memory
  + Each spark executor is preferably allocated a task that requires it to read partition closest to it in network , observing data locality
  + Partitions allow for efficient parallelism



* A distributed scheme of breaking up data into chunks or partitions allows Spark executors to process only data that is close to them, minimizing network bandwidth.



The Developer’s Experience

Apache spark was adopt because of :

* Its easy to use API for operationg on small to large data sets acess language Scala Java Python SQL and R

Who Uses Spark, and for What?

* Developer who grapple with big data ( data engineers, data scientist or machine leanring engineers)
* Allow to build a range of applications using a single engine, familiar with their programming languages

Data science tasks

* Cleanse data , explore it to discover patterns and builfs models to predict or suggest outcomes
* Their use SQL , Numpy, pandas, R and python
* Spark MLlib offers a common set of machine learning algorithms to build pipelines, Spark SQL and Spark shell facilitae interactive and ad hoc exploration of data
* Spark 2.4 introduced a new gang scheduler, as part of project Hydrogen to accommodate the fault tolerant need of training and Spark 3 introduce the ability to support GPU resource collection (for deep learning)

Data engineering tasks

* Deploying model
* Build and transform raw, dirty data into clean data that is easly consumable or usable by other data scientist
* Build large pipeline for with streaming engine / with web application
* Performance improvement on Spark 2.x and 3.0 due to Catalsyt optimizer (for SQL and Tungsten (for compact code généretation

Popular Spark use cases

* Whether you are a data engineer, data scientist, or machine learning engineer, you’ll
* find Spark useful for the following use cases:
* Processing in parallel large data sets distributed across a cluster
* Performing ad hoc or interactive queries to explore and visualize data sets
* Building, training, and evaluating machine learning models using MLlib
* Implementing end-to-end data pipelines from myriad streams of data
* Analyzing graph data sets and social networks
* Spark 1.0 first released 2014

CHAPTER 2

Downloading Apache Spark

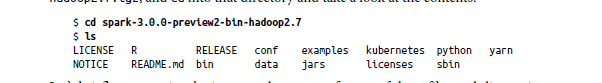
and Getting Started

* In local mode running, procession is donne on a single machine, spark shell which providing quick feedback loop for iteratively performing spark operations
* Spark shell help to prototype spark operation with small dara sets before writing a complex spark application
* Spark shell only supports scala , python,and R

Step 1: Downloading Apache Spark

* Since Spark 2.2, when we want only use Pyspark we can install through PyPI repository
* Some extra dependencies can be created pip install pyspark[sql,ml,mllib] (or pip install pyspark[sql] if you only want the SQL dependencies).

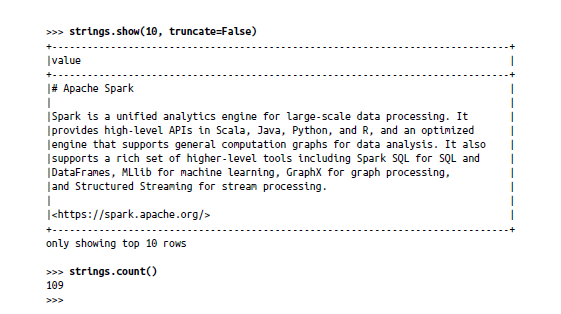
Spark’s Directories and Files



* Bin
  + Contains most of the scipts employ to interact with spark (spark-shell, spark sql, pyspark, sparkR)
  + Can submit standalone application using spark-submit
  + Scripts to builds and pushes docker images when running spark with kubernetes support
* Sbin
  + Script with administrative purpose for starting and stopping spark components in the cluster in its various deployment modes
* Kubernetes
  + Contains dockerfiles for creating docker images for spark on a kubernetes cluster
  + Contains fole providing instructions on how to build the spark distribution before building your docker images
* Data
  + Directory populated with \*.txt files that serve as input for spark’s components: MLlib, structured streaming, and GraphX
* Examples

Step 2: Using the Scala or PySpark Shell

* Spark comes with four widely used interpreters that act like interactive shells and enable ad hoc data analysis: pyspark, spark-shell, spark-sql and sparkR
* Shell has benn augmented to support connecting in the cluster and to allow you to load distributed data into spark wokers memory



Every computation expressed in high-level Structured APIs is

decomposed into low-level optimized and generated RDD operations and then converted into Scala bytecode for the executors’ JVMs. This generated RDD operation code is not accessible to users, nor is it the same as the user-facing RDD APIs.

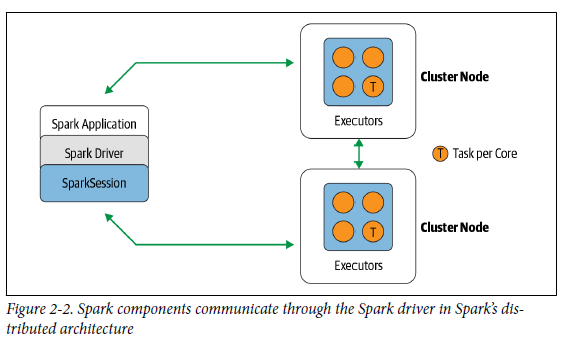
Step 3: Understanding Spark Application Concepts

How spark transformed code and executed as tasks across the spark executors

* Application
  + A user program built on spark using its APIs. Its consists of a driver program and executor on the cluster
* SparkSession
  + An object that provides a point of entry to interact with underlying spark functionality and allows programming spark with its APIs. In an interactive spark shell, the spark driver instantiates a SparkSession
* Job
  + A parallel computation consisting of multiple tasks that gets spawned in response to a spark action (save(), collect())
* Stage
  + Each job gets divied into smaller sets of tasks called stages that depend on each other.
* Task
  + A single unit of work or execution that will be sent to a spark executor

Spark Application and SparkSession

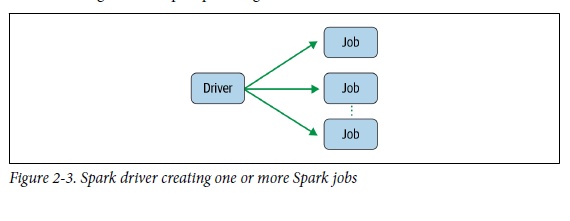
* Spark driver program is at the core of every spark application
* Spark driver created => sparkSession object



* Once you have a SparkSession, you can program spark using the APIs to perform spark operations

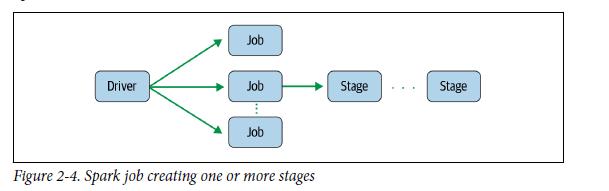
Spark Jobs

* During interactive sessions with spark shells, the driver convert spark application into one or more spark jobs
* Jobs It then => transforms each job into DAG (essence of spark execution plan)
* Each node within a DAG could be a single or multiple spark stages



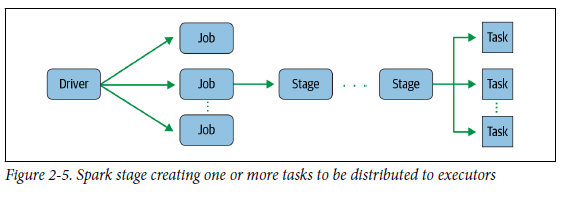
Spark Stages

* Stage are created based on what operations can be perfomed serially or in parallel (not all spark operations can happen in a single stage)
* Spark stages delineated on the operator’s computation boundaries, where they data transfer among spark executors ( shuffle)



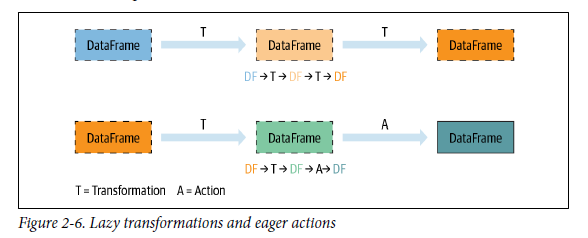
Spark Tasks

* Stage comprised of spark tasks (a unit of execution), federated across each spark executor
* Each task maps to a single core and works on a single partition of data
* with 16 cores can have 16 or more tasks working on 16 or more partitions in parallel, making the execution of Spark’s tasks exceedingly parallel!

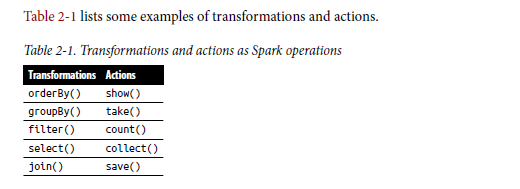


Transformations, Actions, and Lazy Evaluation

* Spark Operations is classified into two types : transformations and actions
* Transformations
  + Transform spark dataframe into new dataframe without altering the original data it the prooerty of immitability
  + Transformations will not change the original dataframe , instead it will retrun the transformed results of the operation as new dataframe
  + All transformation are evaluated lazily (their results are not computed immediately, but they are recorded or remembered as a lineage )
  + Recorded lineage allowd spark to rearrange certain transformations coalesce them or optimize transformation into stage for more efficient execution
  + Lazy evaluation is Spark’s strategy for delaying until an action is invoked or data touched (read or write



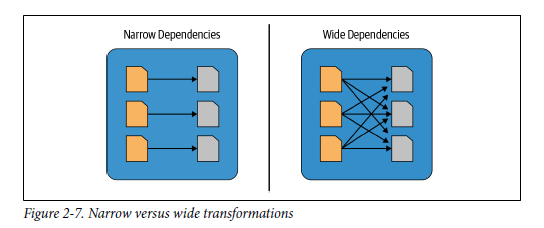
* Lazy evaluation allows spark to optimize your queries by peeking into your chained transformations, lineage and data immutability provide fault tolerance
* Spark record each transformation in its lineage and the DataFrames are immutable between transformations, it can reproduce its original state by simply replaying the recorded lineage giving resiliency



* The actions and transformations contribute to a spark query plan
* The actions triggers the execution of all transformations recorded as part of the query execution

Narrow and Wide Transformations

* Spark transformatons are operations that spark evaluates lazily
* Avatange of lazy evaluation scheme is that spark can inspect the computational query and ascertain how it can optimizer it
* Optimization can be done by joining or pipelining some operations and assigning them to a stage
* Operations is break into Stages by determining which operation require shuffle or exchange of data across clusters
* Transformation can be classify
  + Narrow dependencies: transformation where a single output partition can be computed from a single input partition   
    ex: filter() & contains() are narrow because they can operate on a single partition and produce the resulting output partition without any exchange data
  + Wide dependencies: data from other partitions is read in, combined and written to disk, these force shuffle of data from each of the executor’s partitions across the cluster , require data from other partition for computation

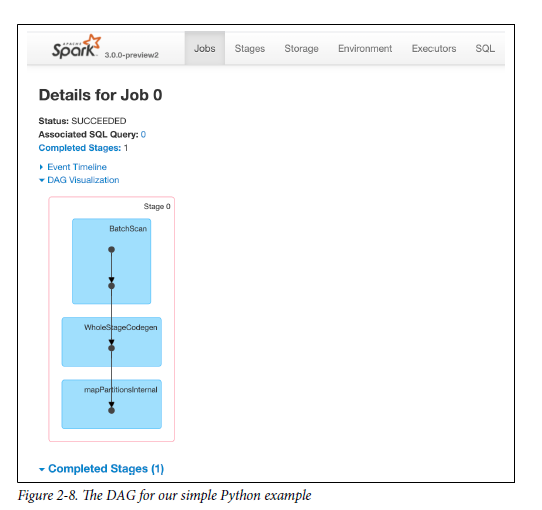


The Spark UI

* Spark include a graphical user interface, it use to
  + Inspect or monitor spark application in their various stage or decomposition (jobs, stages, tasks)
* Driver launches(depending on how spark is deployed) a web UI, running by default on port 4040
  + A list of scheduler stages and tasks
  + A summary of RDD size and memory usage
  + Informations about the environment
  + Informations about the running executors
  + All the spark SQL queries
  + Can be access in local through *http://<localhost>:4040*

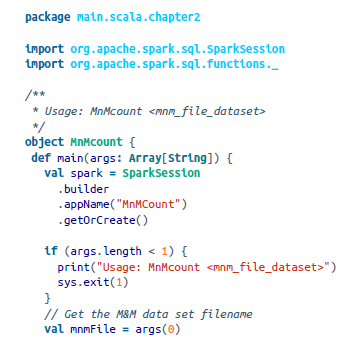
When you launch spark-shell, part of the output shows the localhost

URL to access at port 4040



* mapPartitionInternal for internal moving of data

Your First Standalone Application





Building Standalone Applications in Scala

Because Python is an interpreted language and there is no such

step as compiling first (though it’s possible to compile your Python

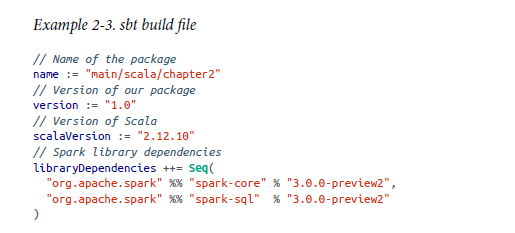
code into bytecode in .pyc), we will not go into this step here. For

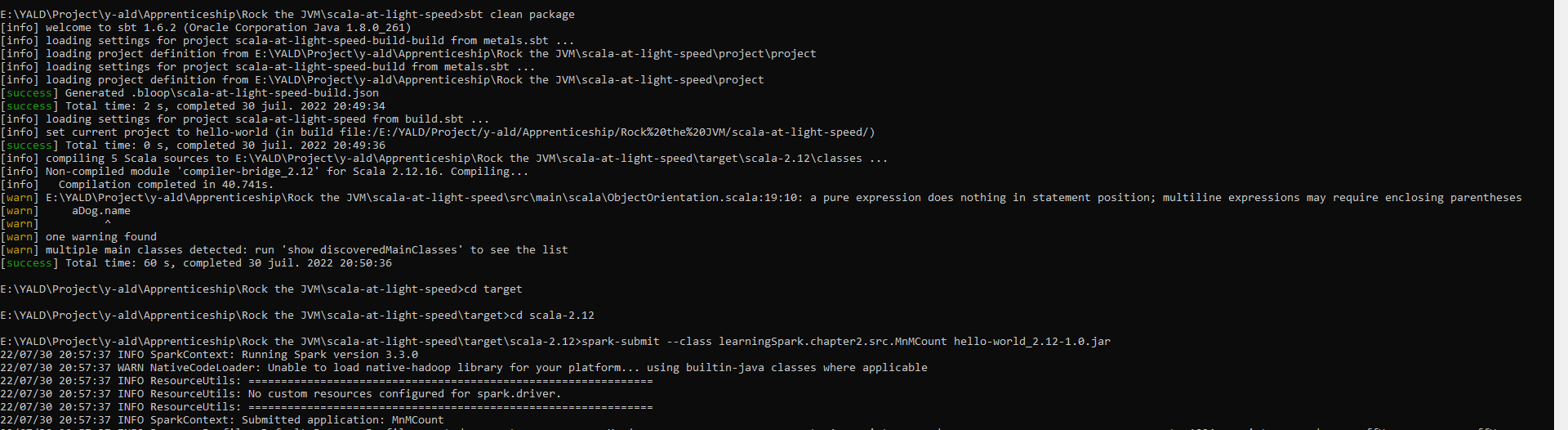
details on how to use Maven to build Java Spark programs, we refer

you to the guide on the Apache Spark website. For brevity in this

book, we cover examples mainly in Python and Scala.

* build.sbt is the specification file that, loke makefile, describes and instruts the scala compler to build yout Scala related tasks (jars, packaches)





spark-submit --class learningSpark.chapter2.src.MnMCount hello-world\_2.12-1.0.jar

CHAPTER 3

Apache Spark’s Structured APIs

* Dataframe was introduce in Spark 1.3 and Spark SQL in spark 1.x
* Creation of high-level APIs (DataFrame and Dataset) unification in spark 2.x

Spark: What’s Underneath an RDD?

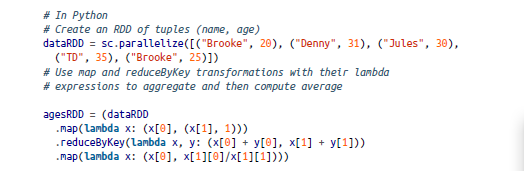
* RDD is the most basic abstraction in spark
  + Dependencies:
    - instructs spark how an RDD is constructed with is inputs is required.
    - When necessary to reproduce results, spark can recreate an RDD from these dependencies and replicate operations on it
    - It gives RDD resiliency
  + Partitions (with some locality information)
    - Partition provide the ability to split the work to parallelize computation on partitions across executors
    - In some case (HDFS) spark will use locality information to send work to executors close to the data (data less transmited over network)
  + Compute function: partition => Iterrator[T]
    - Compute function that produces an Iterator[T] for data that will be stored in the RDD
* There is some problem with these principle
  + Compute function(computation) is opaque to spark, spark see only lambda expression
  + Iterator data type is also opaque for python RDDs, spark only knows that it is a generic object python
  + As spark is unable to inspect the computation or expression , spark has no way to optimize it
  + Spark have no information on data type of, all it can do is serialize the opaque object as a series of bytes , without data compression technique

Structuring Spark

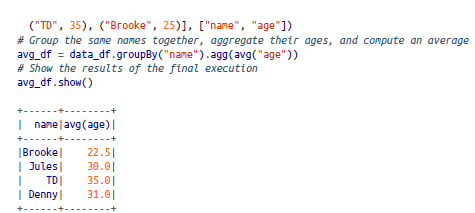
* Spark 2.x introduced a few key schemes for structuring spark
  + Express computation by using common patterns found in data analysis (these pattern are expressed as high-level operations ex: filtering …)
  + Bring clarity and simplicity
* This specificity is further narrowed through use of set of commend operators in a DSL, available as APIs in Spark’s supported language
* These operator tell spark what you wish to compute with your data, the result can construct efficient query plan for execution
* Final scheme of order and structure is to allow to arrange data in tabular format as SQL, with supported structured data types

Key Merits and Benefits

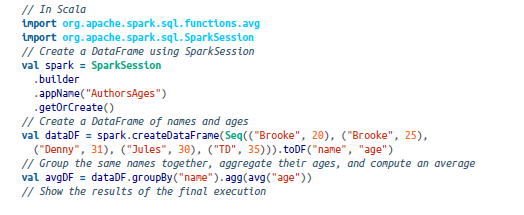
* Structure yields a number of benefits
  + Better performance and space efficiency across spark components
  + Expressivity, simplicity, composability, and uniformity
  + Ex expressivity and composability







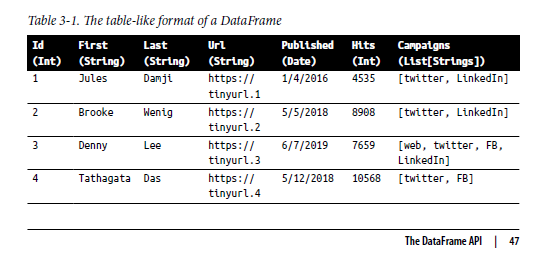
* This code is far more expressive as well as simpler than the earlier version ( because of high-level DSL operator and APIs to tell Spark what to do)
* Use Operator to compose our query and spark API allow spark to understand it
* Spark can inspect and parse these query => can optimize to arrange the operations for efficient execution
* DSL operators introduce to Order and structure by mapped to command recurring data analysis
* Limit the scope of developer to instruct the compiler or control how their queries should e computed



* The simplicity and expressivity allow by Spark SQL engine upon which the high-level structured APIs are builts ( perform query against dataframe in Structured streaming // or MLlib) it uniformize

The DataFrame API

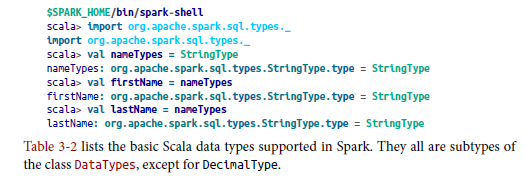
* DataFrame is inspired by pandas DataFrame in structure, format and a few specific operations
* Spark dataFrame are liker distributed in memory tables with named columns and schemas , where each column has a specific data type : integer, string , array map, real date, timestamp etc
* Example of DataFRame in human eye

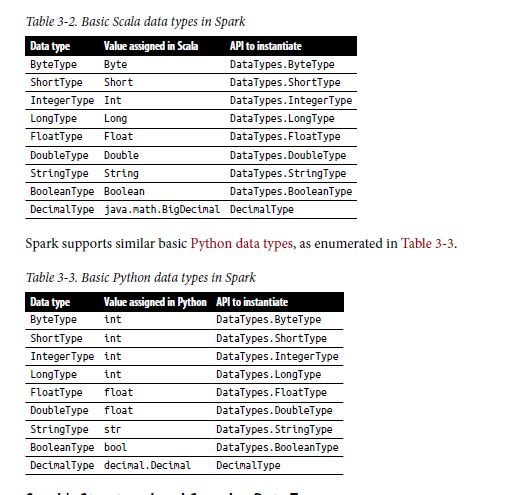


* Put data as structured table
  + Easier to digest and work with when come common operations to execute on row and columns
* DataFrame are immutable and Spark Keeps a lineage transformations
  + Can do various transformations while the previous version are preserved
* A named column in DataFrame and its associated Spark data type can be declared in te schema

Spark’s Basic Data Types

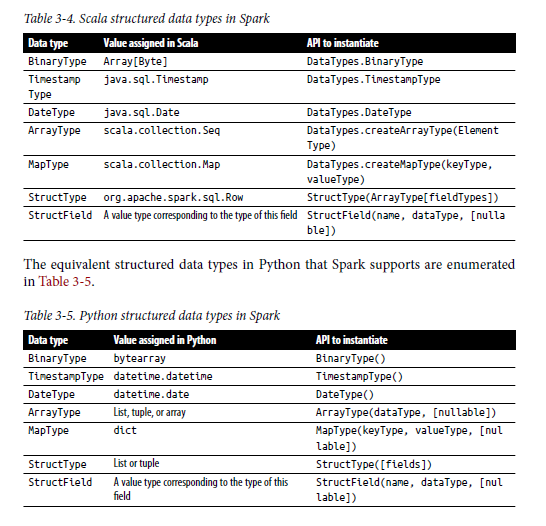
* Matching its supported programming languages, spark supports basics internal data type, which can be declared in spark application or defined in the schema





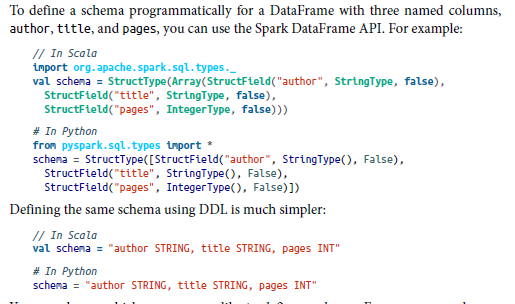
Spark’s Structured and Complex Data Types

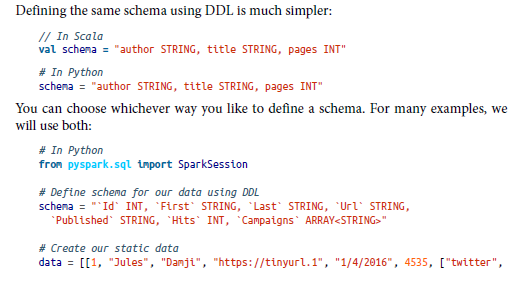
* For complex data analytics as structured or nested we can need to handle these complex data types : maps , arrays, structs, dates , timestamps , fields



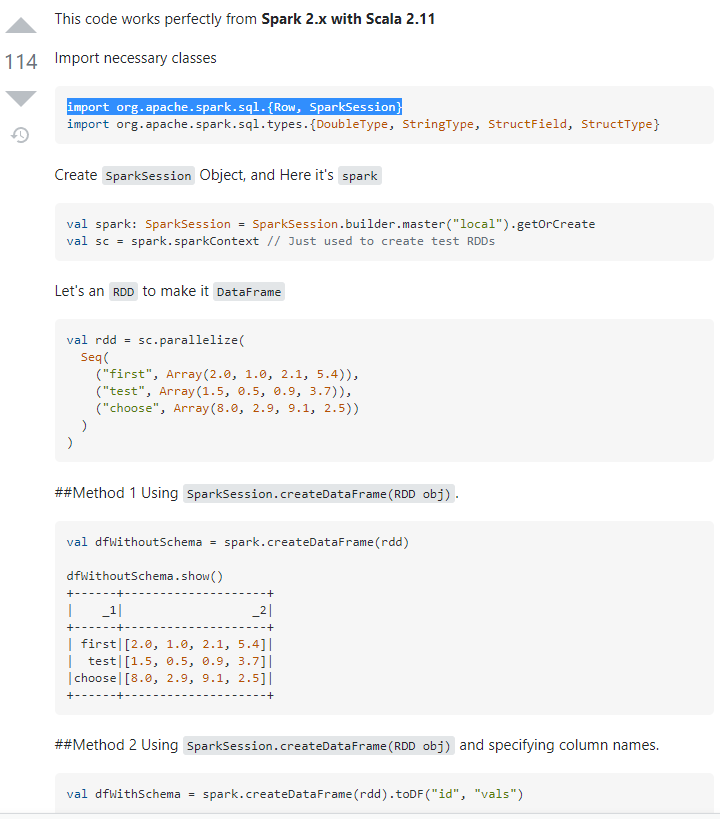
Schemas and Creating DataFrames

* A Schema in spark defines the column names and associated data types for a DataFrame.
* Most often, it use when when reading structured data from an external data source
* Defining a schema up front as opposed to taking a schema on read approach offers three benefits :
  + You relieve spark from the onus of inferring data types
  + You prevent spark from creating a separate job just to read a large portion of your file to ascertain the schema, which for a large data file can be expensive and time-consuming
  + You can delect errors early if data doesn’t match the schema
* Define schema



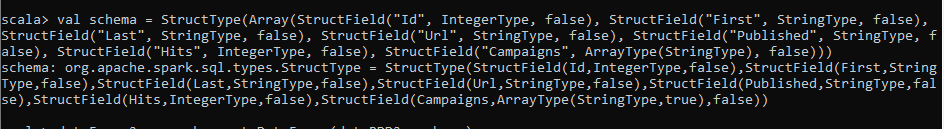


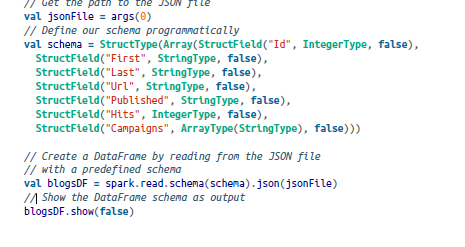












import org.apache.spark.sql.types.\_

import org.apache.spark.sql.Row

val schema = StructType(Array(StructField("Id", IntegerType, false), StructField("First", StringType, false), StructField("Last", StringType, false), StructField("Url", StringType, false), StructField("Published", StringType, false), StructField("Hits", IntegerType, false), StructField("Campaigns", ArrayType(StringType), false)))

val dataRDD2 = sc.parallelize(List((1, "Jules", "Damji", "https://tinyurl.1", "1/4/2016", 4535,List("twitter","LinkedIn")),(2, "Brooke","Wenig", "https://tinyurl.2", "5/5/2018", 8908, List("twitter","LinkedIn"))))

val dataFrame2 = spark.createDataFrame(dataRDD2.map(x => Row(x.\_1, x.\_2, x.\_3, x.\_4, x.\_5, x.\_6, x.\_7)), schema)

Columns and Expressions

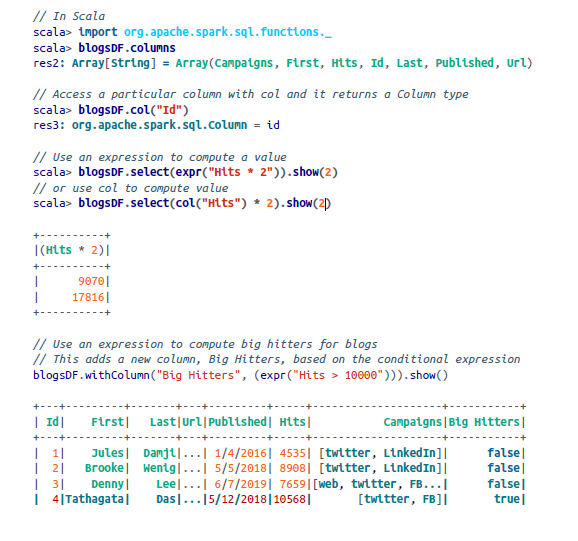
* Named columns in DataFrame are conceptually similar to named columns in pandas or R DataFrames or in an RDBMS
  + They describe the a type of field
  + All column can be listed by they name
  + Operation on value of the column using relational and computational expressions
* Columns are objects with public methods (represented by column type)
* Logical and mathematical expression can be used on column EX: expr("columnName \* 5") or(expr("columnName- 5")>col(anothercolumnName)),
* Expr() is part of org.apache.spark.sql.functions (Scala) packages

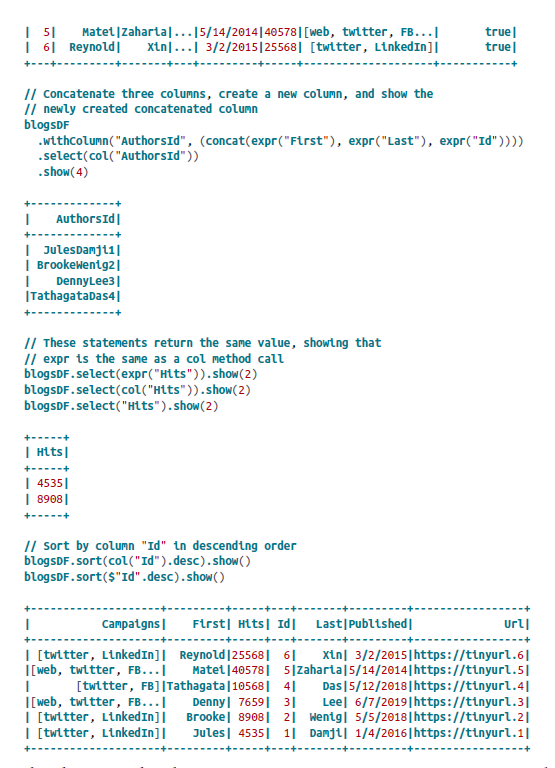
Scala, Java, and Python all have public methods associated with columns.

You’ll note that the Spark documentation refers to both col

and Column. Column is the name of the object, while col() is a standard

built-in function that returns a Column.





* In this last example, the expressions blogs\_df.sort(col("Id").desc) and blogsDF.sort($"Id".desc) are identical. They both sort the DataFrame column named Id in descending order
  + Explicit function col(“Id”) to return a column object
  + $ convert column name into Column object

We have only scratched the surface here, and employed just a couple of methods on Column objects. For a complete list of all public methods for Column objects, we refer you to the Spark documentation.

* Column object in a DataFrame can’t exst in isolation; each column is part of a row in a record and all the rows together contitue a DataFrame

Rows

* A Row in spark is a generic Row object, containing one or more columns (same or different type)
* Row can be instantiate in each of spark’s supported language, cause is an object in spark and ordered collection of field



* Can be used to created dataFrame

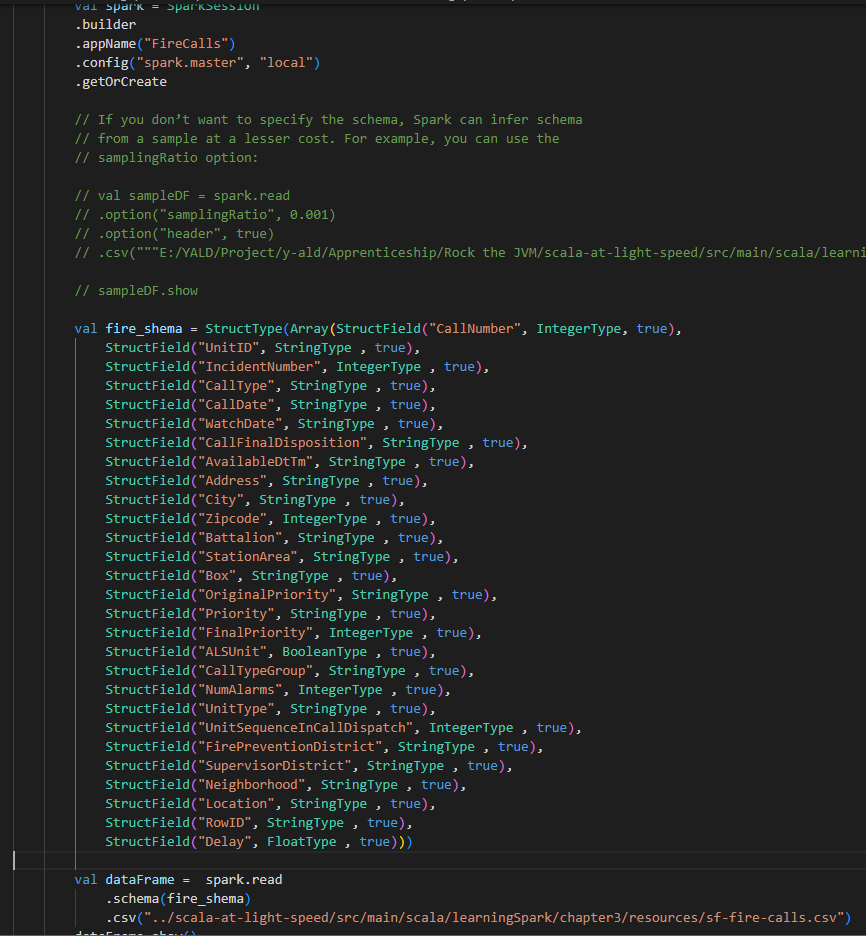


Common DataFrame Operations

* To perform common data operations on DataFrames , need to load a DataFrame from a dara source that holds your structured data
  + DataFrameReader : enable to read data to DataFrame from different data sources format JSON, CSV, parquet, Avro, ORC, Text …
  + DataFrameWritrer to write back DataFame to different data sources format

Using DataFrameReader and DataFrameWriter

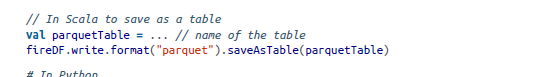
* Reading and writing are simple because of hilgh level abstraction
* Data source include NoSQL sotres, RDBMS, streaming engine Apache Kafka and Kinesis



* For writing data parquet is the most popular format and it is the default one
  + When writing in parquet it keeps the schema in a metadata file, so no need to specific the schema in cas of subsequent reads
* Save as parquet or SQL table

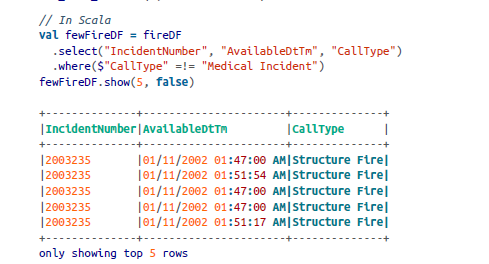


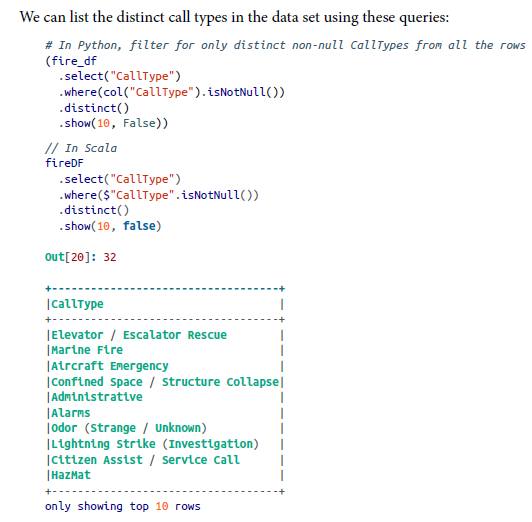
* When save as table it registers metadata with the Hive metaStore



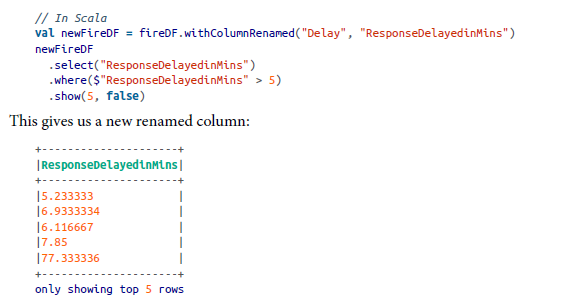
Transformations and actions

* Projections and filters.
  + Projection (in relational parlance) is a way to return only rows matching a certain relational condition by using filters
  + In spark is done with select(), filter(), where()

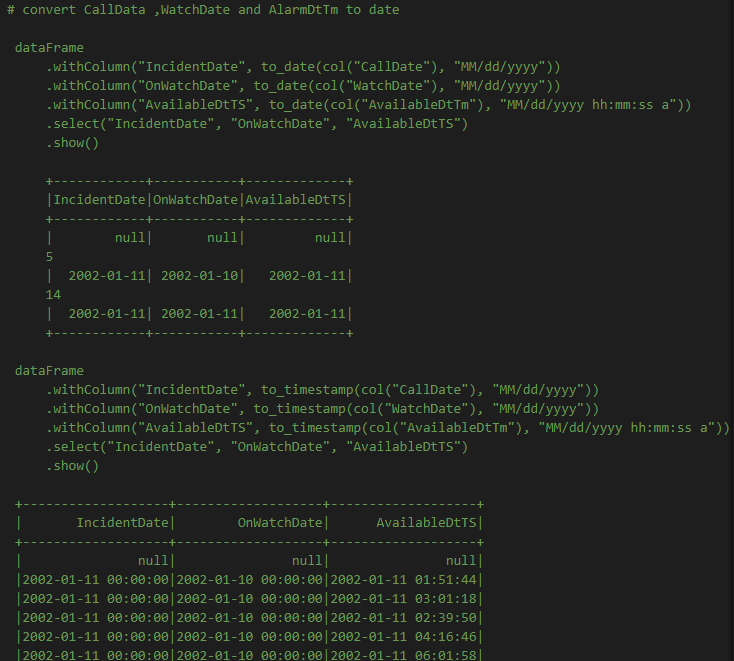




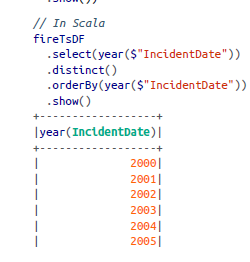
* Renaming, adding, and dropping columns.
  + Space in column can be problematic spcially when try to write in some data source (parquet)



Because DataFrame transformations are immutable, when we rename a column using withColumnRenamed() we get a new DataFrame while retaining the original with the old column name.



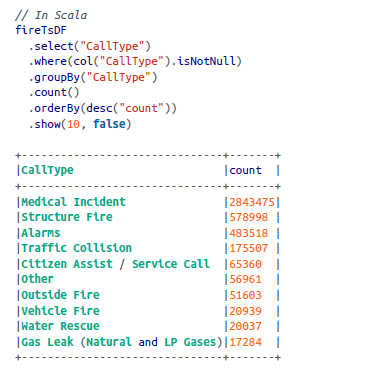
* Another transformation date , leet see what it do
  + Convert existing column’s data tume from string to spark supported timestamp
  + Use the new format specified in the format string “MM/dd/yyy” or
  + After converting to the new data , append the new column specified in the argument withColumn
  + Assign the new modified DataFrame to fire\_ts\_df
* Note that there is functions from spark.sql.functions like month(), year(), day() to explore date data EX:



Aggregations. : The purpose of aggregation is to regroupind data by value to simplify the counting, projection, aggregation and filtering

* groupBy(), orderBy() and count() (transformation and action) offer the ability to aggregate by column names and then aggregate counts across them

For larger DataFrames on which you plan to conduct frequent or repeated queries, you could benefit from caching. We will cover DataFrame caching strategies and their benefits in later chapters.



The DataFrame API also offers the collect() method, but for

extremely large DataFrames this is resource-heavy (expensive) and

dangerous, as it can cause out-of-memory (OOM) exceptions.

Unlike count(), which returns a single number to the driver, col

lect() returns a collection of all the Row objects in the entire Data‐

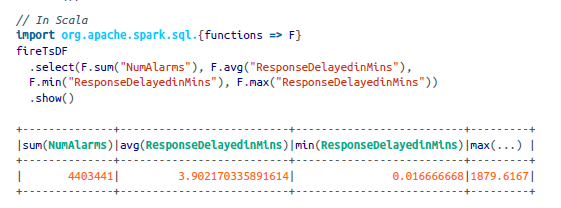
Frame or Dataset. If you want to take a peek at some Row records

you’re better off with take(n), which will return only the first n

Row objects of the DataFrame.

Other common DataFrame operations. :

* DataFrame also provide statistica methods like min(), max(), sum() and avg()
* Some of alarls , the average respond time and minimum maximm respone times



* For more advanced statistical needs there is methods like stat(), describe, correlation, covariance, sampleBy, approxQuantile, frequentItems and so on

End-to-End DataFrame Example

TODO

• What were all the different types of fire calls in 2018?

• What months within the year 2018 saw the highest number of fire calls?

• Which neighborhood in San Francisco generated the most fire calls in 2018?

• Which neighborhoods had the worst response times to fire calls in 2018?

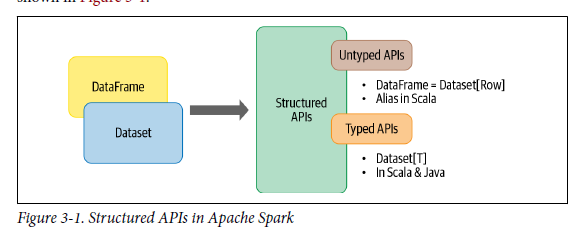
• Which week in the year in 2018 had the most fire calls?

• Is there a correlation between neighborhood, zip code, and number of fire calls?

• How can we use Parquet files or SQL tables to store this data and read it back?

The Dataset API

* Spark 2.0 unified the DataFrame and Dataset APIs as Structured APIs with similar interfaces so that developers would only have to learn a single set of APIs
* Datasets take on two characteristic : typed and untyped APIs

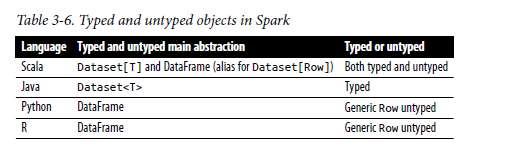


* Conceptuality DataFrame in Scala is an alias for a collection of generic object, Dataset[Row] (Row is a generic untyped JVM object that may holds different types of fields )
* Dataset is a collection of strongly typed JVM objects in Scala or a class in JAVA

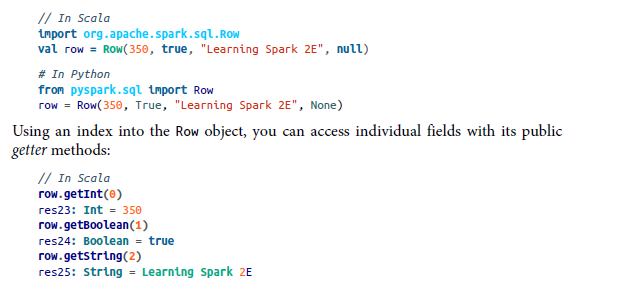
a strongly typed collection of domain-specific objects that can be transformed in parallel using functional or relational operations. Each Dataset [in Scala] also has an untyped view called a DataFrame, which is a Dataset of Row.

Typed Objects, Untyped Objects, and Generic Rows

* Datasets make sense only in scala and java (cause python and R are not compile time type safe, types are dynamically inferred or assigned during execution)
  + In Scala, java types are bound to variable and object at the compile time



* Row is a generic object type in spark, holding a collection of mixed types thanc ca be accessed using an index.
  + Internally spark manipulates row as objects , converting them to the equivalent types covered

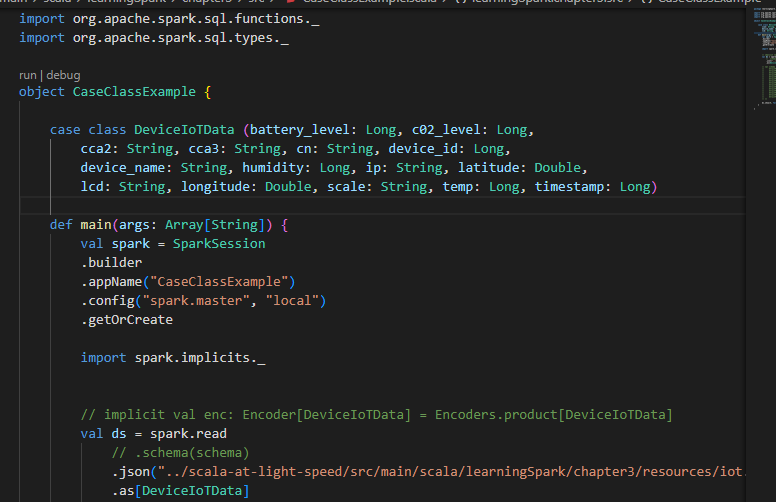


* Each element in a dataset maps to JVM object

Creating Datasets :

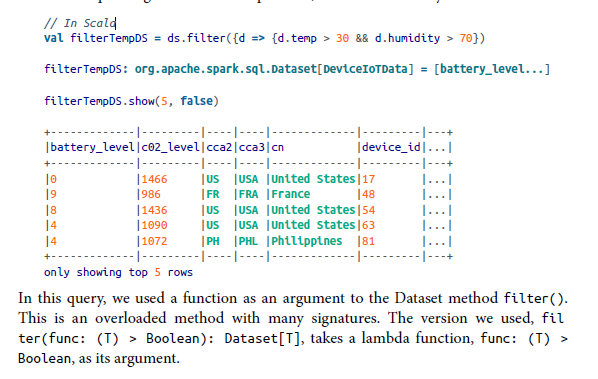
* As for creating Dataframe we should know schema when creating datasets (know the data types)
  + Possible to infer schema with CSV and JSON ( consume resource for large datasets )
* In scala the easiest way is to use case class // in Java JavaBean class are used

Scala: Case classes

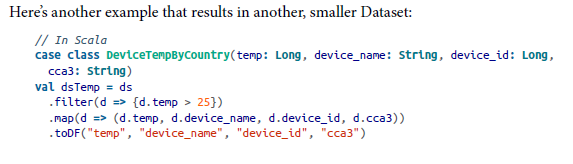
* 

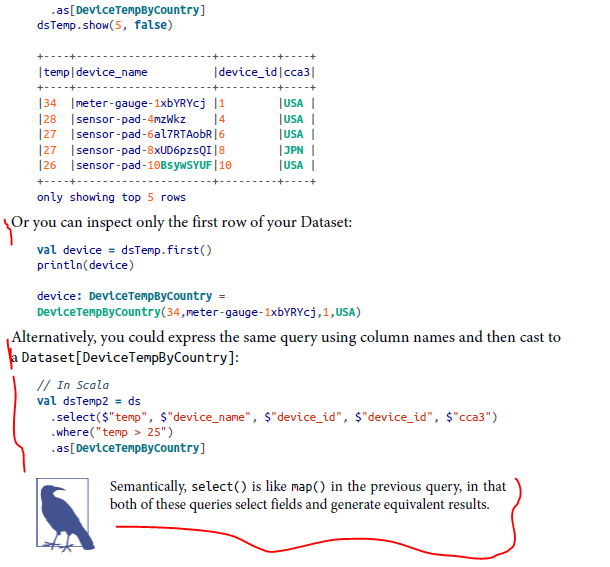
Dataset Operations

* We can also perfom transformation and action on Datasets
* Dataset Takes lambda function



* The argument of the lambda function is a JVM object of type DeviceIoTData
  + As such can acces its fields using the dot (.)
* With DataFrame we express filter condition as SQL-like DSL operation, which are language agnostic
  + With dataset we use language-native expression as Scala or java code





* Operations we can perform on Dataset (filer, map, groupBy, select, take etcs
  + Are similar to ones in Dataframes
  + Dataset is similar to RDD caused they provide similar interface to its aforementioned methods and compile time safety vut with a much ieasier to read and an object oriented programming interface
* When using dataset Spark SQL engine
  + Handles the creation n conversion, serialization and deserialization of the JVM objects
  + Care of java heap memory management with the help of dataset encoders

End-to-End Dataset Example

1. Detect failing devices with battery levels below a threshold.

2. Identify offending countries with high levels of CO2 emissions.

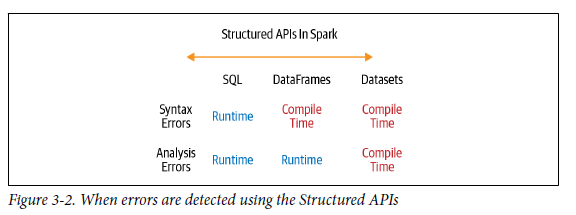
3. Compute the min and max values for temperature, battery level, CO2, and

humidity.

4. Sort and group by average temperature, CO2, humidity, and country.

DataFrames Versus Datasets

* When use DataFrame or Datasets can depend of the language we want use
* If you want tell spark what to do and not how to do it use DataFame or Datasets
* If yo want rich semantics, high level abstraction and DSL operators use Dataframes or data sets
* If you want strict compile time type safety and don’t mind createing multiple case class for a specific Dataset[T], use dataset and dataframe)
* If processing demands high level expressions (filter, maps computing average or queries
* If your processiong dictates relation transformations similar to SQL like queries, use dataframe
* If you want to take avantage of benefit from tugsten’s efficient serial
* If yiou wnant unification, code optimization , and simplifications of APIs across spark components , use dataframe
* If you are an R user, use DataFrames.
* If you are a Python user, use DataFrames and drop down to RDDs if you need more control
* If you want space and speed efficiency, use DataFrames.
* If you want errors caught during compilation rather than at runtime, choose the appropriate API as depicted in

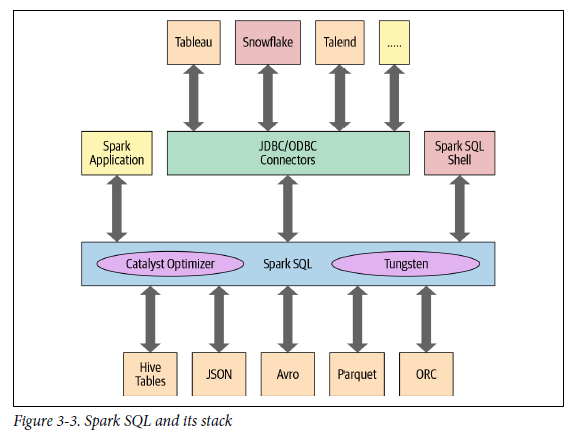


When to Use RDDs

* Using Third party package that’s writing using RDDs
* Can forgo the code optimization, efficient space utilization and performance benefits available with dataframe and datasets
* Want to precisely instrucr spark how to do a query
* Can move from DataFrame or Datasets to RDDs using (df.rdd )
* Dataframe and Dataset are built on top of RDD and decomposed to compact RDD code during whole stage code generation

Spark SQL and the Underlying Engine

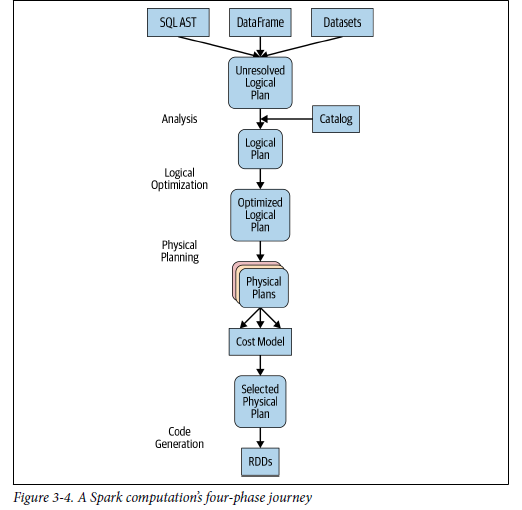
* Spark SQL allow developers to issue ANSI SQL:2003(compactible queries on structured data with a schema) at programmatic level
* Spark SQL engine (Spark 3):
  + Unifies Spark components and permits abstraction to DataFrames/Datasets in Java, Scala, Python, and R, which simplifies working with structured data sets.
  + Connects to the Apache Hive metastore and tables.
  + Reads and writes structured data with a specific schema from structured file formats (JSON, CSV, Text, Avro, Parquet, ORC, etc.) and converts data into temporary tables.
  + Offers an interactive Spark SQL shell for quick data exploration.
  + Provides a bridge to (and from) external tools via standard database JDBC/ODBC connectors.
  + Generates optimized query plans and compact code for the JVM, for final execution.



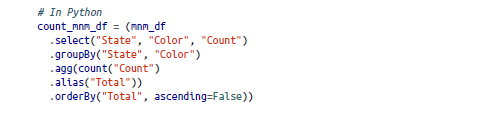
* At the core of the spark engine
  + Catalysy Optimize
  + Project Tungsten

The Catalyst Optimizer

* Catalyst optimizer takes a computational query and converts it into execution plan, it goes through
  + Analysis
  + Logical optimization
  + Physical planning
  + Code generation

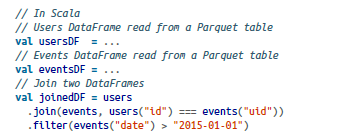


* EX



* To see plan explain df.explain python // df.queryExecution .logical or df.queryExecution.optimizedPlan (scala)







* Phase 1: Analysis
  + Spark SQL engine begins by generating an abstract syntax tree (AST) for SQL or DataFRame query
  + Here any columns or table names will be resolved by consulting internal catalog (a programmatic interface to Spark SQL that holds a list of names of columns, data types, function, tables , database)
* Phase 2: logical optimization, is composed of 2 phases
  + Applying a standard rule based optimization approach, it will first construct a set of multiple plan
  + using its cost based optimize (CBO), assign cost to each plan
  + these plan are laid out as operator trees, the may include
    - the process of constant folding, predicate pushdown, projection pruning, Boolean expression simplification
* phase 3: physical planning
  + Generate optimal physical plan for the selected logical plan using physical operators
* Phae 4: Code generation
  + Generating efficient java bytecode to run on each machine
  + Can be use **state of the art compiler technology** for code generation due to the fact that spark can operate data in memory to speed execution (acts as a compiler )
  + Project tungsten which facilitates whole stage code generation plays a role here
    - Whole stage code generation : physical query optimization phase that collapses the whole query into a single function, getting rid of virtual function calls and employing CPU register for intermediate data
    - In spark 2.0 the second generation Tungsten engine use this approach to generate compact RDD code for final uexcutin (CPU efficiency and performance)

I

We have talked at a conceptual level about the workings of the

Spark SQL engine, with its two principal components: the Catalyst

optimizer and Project Tungsten. The internal technical workings

are beyond the scope of this book; however, for the curious, we

encourage you to check out the references in the text for in-depth

technical discussions.

CHAPTER 4

Spark SQL and DataFrames:

Introduction to Built-in Data Sources

Using Spark SQL in Spark Applications

* The sparksession introduced in spark 2.0 provides a unified entry point for programming spark with the structured APIs
* Sparksession can be used to access spark functionality
* To trait any sql query we use sql() method on the SparkSession instance, spark   
  such as spark.sql("SELECT \* FROM myTableName")

Basic Query Examples

* For enabling structured query (SQL syntax) spark manages all the complexities of creating and managing views and tables both in memory and on disk

SQL Tables and Views

* Tables hold data
  + Associated with each table in spark is its relevant metadata (with these informations)
    - Schema
    - Description
    - Table name
    - Database name
    - Columns names
    - Partitions
    - Physical location where data resides
  + All of this stored in central metastore
* Spark by default use Apache hive metastore (located /user/hive/warehouse)
  + To persit all the metadata about the tables
  + Can change location via config : spark.sql.warehouse.dir

Managed Versus UnmanagedTables

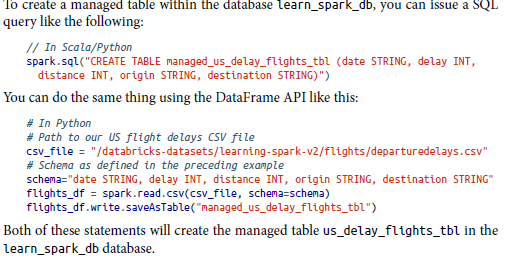
* Spark allow to create 2 type of tables
  + Managed
    - Spark manages both the metadata and the data in the file store
    - This could be a local filesSystem, HDFS or object sotre amazon S3, Azure Blob
    - Drop Table deletes both the metadata and the data
  + Unmanaged
    - Spark manage only the metadata, while external data source such as Cassandra
    - Drop table deletes only metadata

Creating SQL Databases and Tables

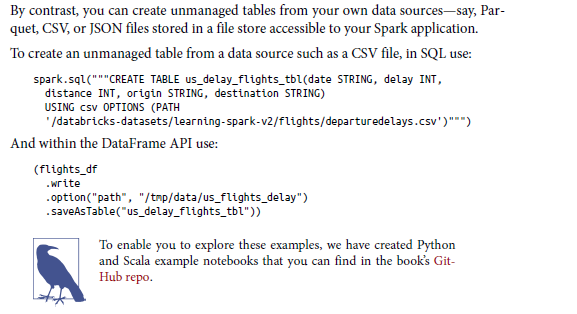
* Tables reside within a database, by default spark recreates tables under the default database

Creating a managed table



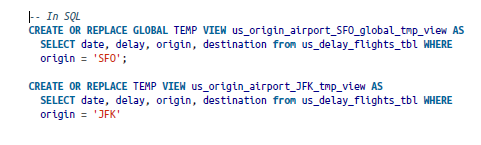


Creating an unmanaged table

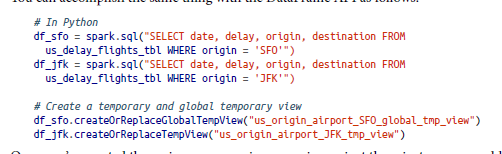


Creating Views

* In addition to creating tables, spark can create views on top of existing tables
* Views can be global(visible across all sparksession on a given cluster) or session scoped (visible only to a single sparksession)
* they temporary they disappear after spark applications terminates
* after create view we can query as it is a table
* difference between a view and table
  + views don’t actually hold the data
  + tables persist after spark application after spark application terminates and views disappear
* create view using SQL from table



* using DataFrame API



* Can queries view as we queries tables
* To access Global temporary we most specify global\_temp.<view\_name> because spark creates global temporary view in global temporary database
  + SELECT \* FROM global\_temps.us\_origin\_airport\_SFO\_global\_tmp\_view
  + SELECT \* FROM us\_origin\_airport\_SFO\_global\_tmp\_view (normal view can be access without prefix)
* Access view
  + Spark.read.table(view\_name)
  + Spark.sql(SELECT \* FROM view\_name)
* We Can Drop view as we Drop table
  + DROP VIEW IF EXISTS us\_origin\_airport\_SFO\_global\_tmp\_view
  + DROP VIEW IF EXISTS us\_origin\_airport\_JFK\_tmp\_view

Temporary views versus global temporary views:

* Temporary view is tied to a single SparkSession within a Spark application
* Global Temporary view is visible across multiple sparksession
* Multiple sparksession can be created in a single spark application (want to access data from different sparksession don’t share the same Hive metastore)

Viewing the Metadata

* Metadata is captured in the Catalog (high level abstraction in spark SQL for storing metadata)
* Catalog functionality was expanded in spark 2.x
  + Spark.catalog.listDatabases()
  + Spark.catalog.listTables()
  + Spark.catalog.listColumns(“us\_delay\_fligts\_tbl)

Caching SQL Tables

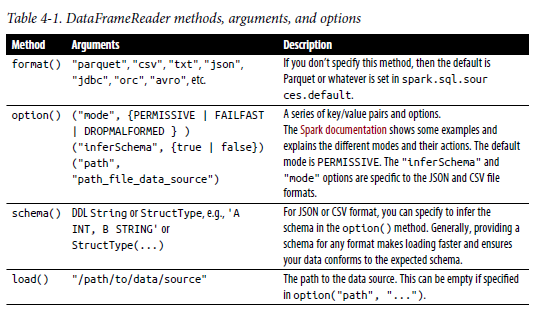
* Can cache and uncached SQL tables and views in Spark 3.0
* Can specify table as lazy ( means you should cached only after the first used instead of immediately)
  + CACHE [LAZY] TABLE table-name
  + UNCACHE Table talbe-name

Reading Tables into DataFrames

* Can query table directly and assign it to DataFrame
  + Val Fligth = spark.sql(“SELECT \* FROM us\_delay\_flights\_tbl”)
  + Val Flight = spark.table(“us\_delay\_flights\_tbl”)

Data Sources for DataFrames and SQL Tables

* Spark SQL provides an interface to a variety of data sources and Set of methods for reading and writing data and from these data source using the Data sources API
* Tow high-level data source API
  + DataFrameReader
    - * Is the core construct for reading data from a data source into a DataFrame
        + DataFrameReader.format(args).option(“key”, “value”).schema(args).load()
        + Stringing methods together
      * Can acess a DataFrameReader only through a SparkSession instance
        + Spark.read: return a handle to DataFrameReader to read into a DataFrame From Static data source
        + Spark.readStream: return an instance to read from streaming source





In general, no schema is needed when reading from a static Parquet

data source—the Parquet metadata usually contains the schema, so

it’s inferred. However, for streaming data sources you will have to

provide a schema.

Parquet is the default and preferred data source for Spark because

it’s efficient, uses columnar storage, and employs a fast compression

algorithm. You will see additional benefits later (such as columnar

pushdown), when we cover the Catalyst optimizer in

greater depth.

* + DataFrameWriter
    - Saves or writes to specified built in data source
    - Access its instance form DataFrame (not sparksession)
    - Pattern
      * DataFrameWriter.format(args)

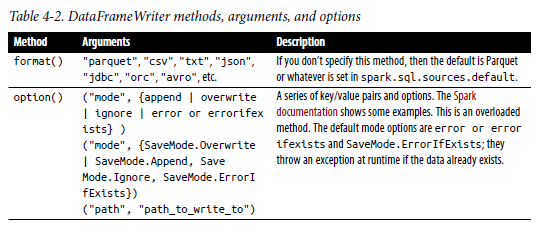
.option(args)

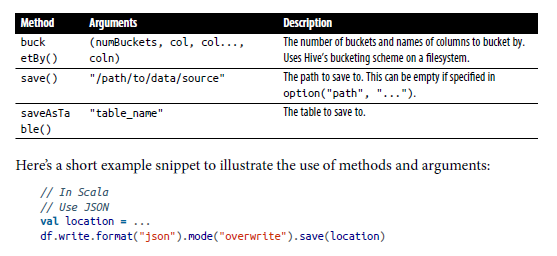
.bucketBy(args)

.partitionBy(args)

.save(path)

* + - * DataFrameWriter.format(args).option(args).sortBy(args).saveAsTable(table)
    - To get an instance handle :
      * Dataframe.write
      * DataFrame.writeStream



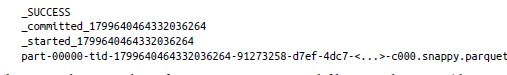


Parquet

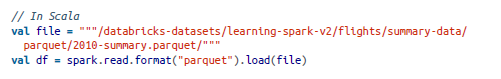
* Parquet is the default data source in spark
* Parquet is an open source columnar file format that offers many I/O optimization (compression, which saves storage space and allow quick access to data columns

Reading Parquet files into a DataFrame :

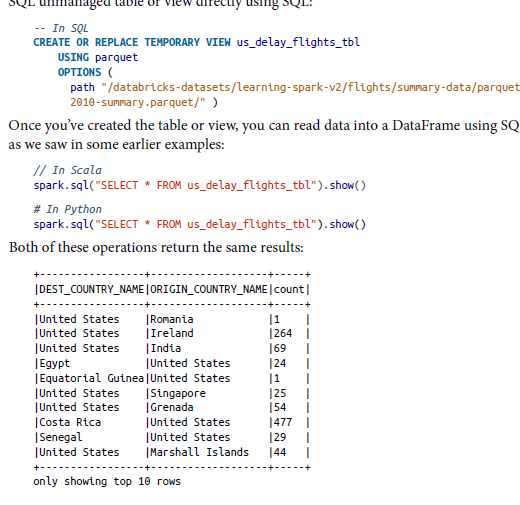
* Parquet are stored in a directory structure that contains the data files , metadata , number of compressed files and some status files
  + Metadata in the footer contains the version of the file format, schema and column data such as the path



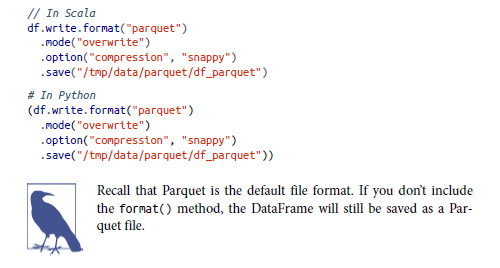
* + Read parquet file just need to specify the schema (unless reading from streaming data no need to supply schema)



Reading Parquet files into a Spark SQL table :

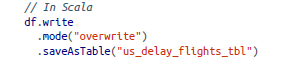


Writing DataFrames to Parquet files :



* This will created a set of compact and compressed parquet files at specific patch
* Snappy is used for compression

Writing DataFrames to Spark SQL tables :

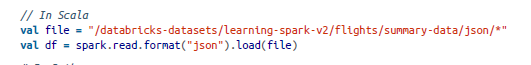


* Use SaveTable instead of save

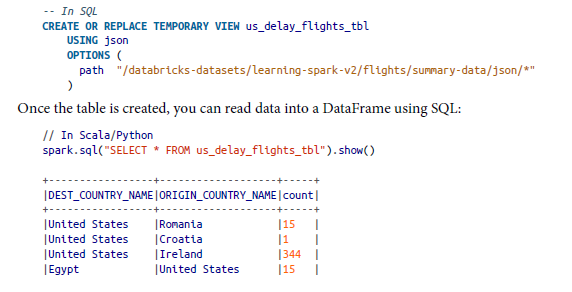
JSON

* JavaScript object Notation (JSON) :
  + Easy to read and easy to parse (compared yo XML)
* Has two representation:
  + Single line mode: each line denotes a single JSON object
  + Multiline mode: entire multiline object constitutes a single JSON object
    - To read it set multiline to true in the option method

Reading a JSON file into a DataFrame :



Reading a JSON file into a Spark SQL table :



Writing DataFrames to JSON files :

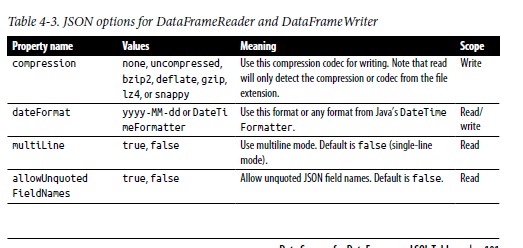
* df.write.format(“json”)

.mode(overwrite)

.option(“compression”, “snappy”)

.save(path)

JSON data source options



CSV :

* text format where filed are delimited by comma (as separator), it can used other separator too

Reading a CSV file into a DataFrame

Val file = “/path”

Val schema = “DEST\_COUNTRY String, ORIGIN\_COUNTRY STRING, count INT”

Val df = spark.read.format(“csv”)

// schema(schema)

.option(“inferSchema’, true)

.option(“header”, “true”)

.option(“mode”, “FAILFAST”) //exit if there is any error

.option(“nullValue”, “”) .. replace any null data with quotes

.load(file)

Reading a CSV file into a Spark SQL table

CREATE OR REPLACE TEMPORATA VIEW us\_delay

USING CSV

OPTION (

path “path/to/data”,

header “true”,

inferSchema “true”,

mode “FAILFAST”

)

Once the table is created we can read into DataFame using SQL

Val df = spark.sql(SELECT \* FROM table\_nae)

Val df = spark.table(table\_name)

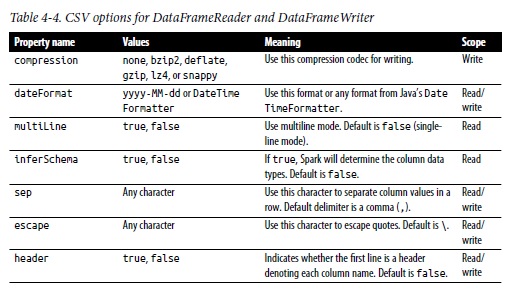
Writing DataFrames to CSV files :

dataFrame.write.format(“csv”)

.mode(“overwrite”)

.save(path/to/data)

CSV data source options



Avro

* Introduced in spark 2.4
* Avro Format is used for example by Apache Kafka for message serializing and deserializing
* Offer direct mapping to JSON speed and efficiency and bindings available for many programming languages

Reading an Avro file into a DataFrame :

Val df = spark.read.format(“avro”)

.load(“path/to/”)

df.show()

Reading an Avro file into a Spark SQL table

CREATING OR REPLACE TEMPORARY VIEW episode\_tbl

USING avro

OPTIONS (

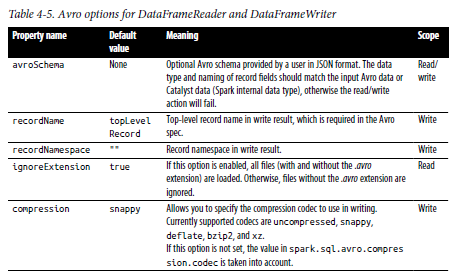
Path “/path/to/”

)

Writing DataFrames to CSV files

Df.write.format(‘avro”).mode(“overwrite”).option().save(“path/to/)

Avro data source options



ORC

* Optimized columnar file format
* Spark 2.x support a vectorized ORC reader
* 2 set configuration dictate which ORC implementation is use
  + Spark.sql.orc.impl set to native
  + Spark.sql.orc.enableVectorizedReader is set to true spark use Vectorized ORC reader
  + Vectorized reader reads blocks of rows (1024 per block) instead of one row at tile
  + It help to reduce CPU usage for intensive operations (scans, filters, aggregations and joins)
* To configure Hive ORC SerDe (serialization and deserialization) USING HIVE OPTIONS (fileFormat ‘ORC) add spark configuration spark.sql.hive.convertMetastoreOrc is set to true

Reading an ORC file into a DataFrame :

Spark.read.format(“ORC”).load(“path/to/file”)

Reading an ORC file into a Spark SQL table

CREATE OR REPLACE TEMPORARY VIEW table\_name

USING ORC

Options ( path path/to/file)

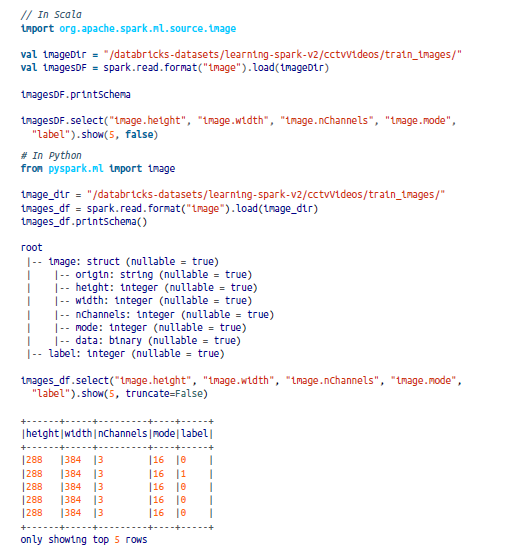
Writing DataFrames to ORC files

Df.write.mode(“overwrite).option(“compression”, “snappy”).save(path/to/file)

Images

* In spark 2.4 images files data sources was introduce (for deep learning and machine learning framework)

Reading an image file into a DataFrame

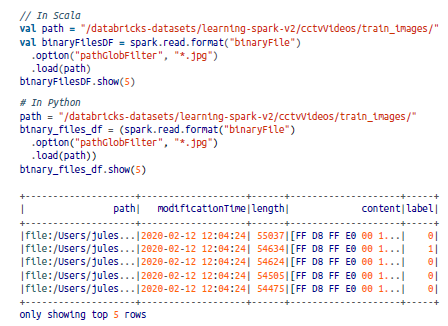


Binary Files

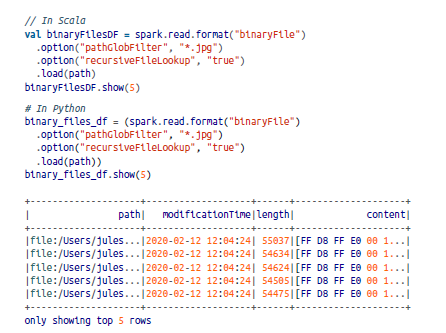
* Binary files as a data source added in spark 3.0
* DataFrameReader convert each binary file into a single DataFRame row(record) content row data and metadata
* It has these columns
  + Path: StringType
  + modifcationTime: TimestampType
  + length: LongType
  + content: BinaryType

Reading a binary file into a DataFrame :

* specfy the data source as a binaryFile
* can be load file matching a given global pattern while preserving the behavior of partition discovery with data source option pathGlobFilter



* to ignore partitioning data discovery in a direction, you can set recursiveFile Lookup to True



* label column is absent when the recursiveFileLookup option is set to trye
* the current binary data source does not support writing a DataFrame back to the original file format

CHAPTER 5

Spark SQL and DataFrames:

Interacting with External Data Sources

Spark SQL and Apache Hive

* Spark SQL integrates relational processing with Spark’s functional programming API
* His genesis was in previous work on shark
* Shark was built on the hive codebase on top of Apache spark and became one of the first interactive SQL query engine on Hadoop Systems
* Give best of both worlds fast as an enterprise data warehouse and scaling as well as Hive/MapReduce
* Spark give avantage of faster performance and relational programming (eg. Declarative queries and optimized storage), as well call complex analytics libraries (eg machine learning)

User-Defined Functions :

* Spark allow data engineer to define their own function too None as user-defined function (UDFs)

Spark SQL UDFs :

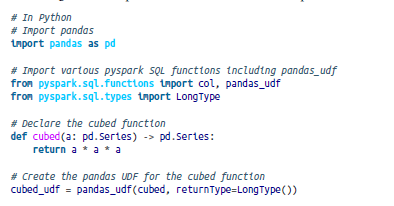
* Benefits of creating an UDFs able to make use within Spark SQL (Ex wrap an ML model within a UDF, for predicting in spark SQL without necessarily understanding the internals of the model)

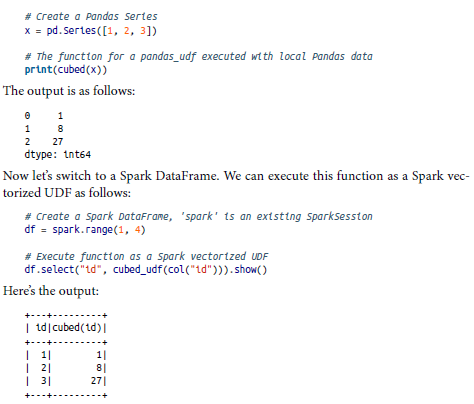
Evaluation order and null checking in Spark SQL :

* Spark Sql include
  + SQL
  + DataFRame API
  + Dataset API
* Spark SQL does not guarantee the order of evaluation of subexpression
  + Exp : spark.sql(‘SELECT s FROM test1 WHERE s IS NOT NULL and srtlen(s)>1”) is not guarantee that the s is not null clause is executed prior to the streln(s)
* To perform proper null checking is recommended to
  + Make UDF itself null aware and do null checking inside the UDF
  + Use IF or CAS WHEN expression to do the null check and invoke the UDF in a conditional branch

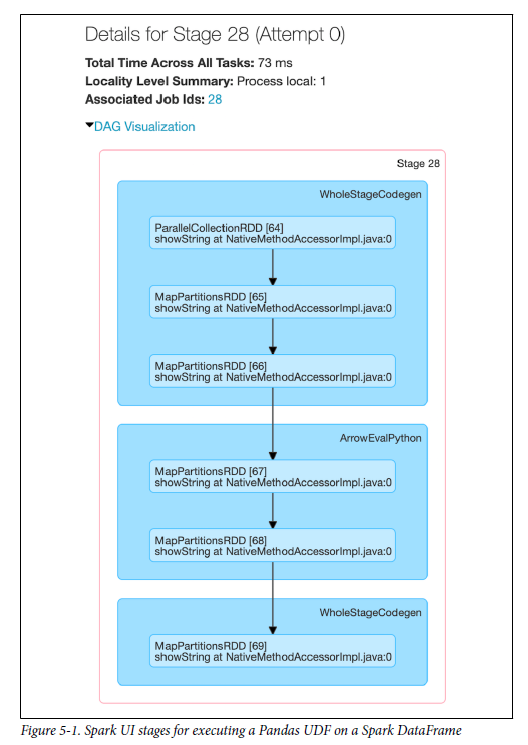
Speeding up and distributing PySpark UDFs with Pandas UDFs

* Previous spark UDFs had slower performance than scala UDFs (caused of movement between the JVM and python)
* To resolve it Pandas UDFs were introduced as part of Apache Spark 2.3
  + Pandas UDF uses Apache Arrow to transfer data and Pandas to work with the data
  + To define panda UDF using the keyword pandas\_udf as decorator or to wrap the function itself
  + Once data is In Apache Arrow format, there is no longer the need to serialize/pickle the data as it is already in format consumable by python process
  + Iis operate on Pandas series or DataFame no individual row
* Since Apace Spark 3.0 pandas UDFs are split into two API categories
  + Pandas UDFs
    - Pandas UDFs infer the pandas UDF type from python type hints in Pandas UDFs such as pandas.Series, pandas.DataFrame, Tuple and Iterator
    - Previously there was a need to define it manually and specify each pandas UDF type
    - Currently supported python type hints in pandas UDFs are series to series, iterator of series to Iterator of series , Iterator of multiple series to Iterator of series and series to scalar (a single value)
  + Pandas Funcrion APIs
    - Pandas Funcrion APIs allow to directly apply local python function to a pyspark dataframe where bot the input and output are pandas instances
    - Supported pandas function API are grouped map, map cogrouped map





* As opposed to a local fnction using vectorized UDF will result in the execution of spark jobs , the firs one exected only on the spark driver (



* + Like many spark job
    - Start with parallelize to send local data (arrow binary batches) to executors
    - call mapPartitions() to convert the Arrow binary batches to spark’s Internal data format (distributed data to spark workers)
    - end uf with WholeStgeCodegen steps, which represent a fundamental step up un performance
    - Pandas UDF is being executed at ArrowEvalPython step

Querying with the Spark SQL Shell, Beeline, and Tableau

Using the Spark SQL Shell

* spark-sql CLI is the convenient tool for executing spark SQL queries
* This Utility cimmunucates wuth the Hive Metasotre service in local mode
* Apache Spark by STS (spark Thrift server) allows JDBC/ODBC client to execute SQL queries over JDBC and ODBC protocols
* To start spark SQL CLI execute this command
  + ./bin/spark-sql
* Once created it can be use to interactively perform Spark SQL queries

Create a table

* Create new table run this queries   
  CREATE TABLE people (name STRING, age int);
* Result locate in /user/hive/warehouse/people
* Insert data into the table  
  INSERT INTO people SLECT name; age FROM …

INSERT INTO people VALUES (“paul”, 10);

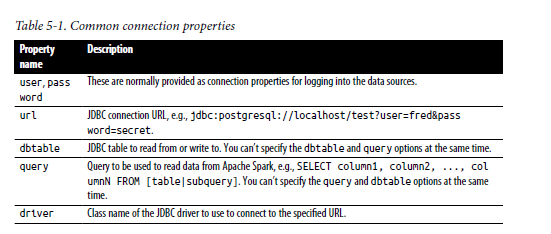
Working with Beeline

* Beeline is a utility for running HiveQL queries against HiveServer2
* Beeline is a JDBC client Based on the SQLLine CLI, JDBC client can be use to execute Spark SQL queries against Spark thrift server

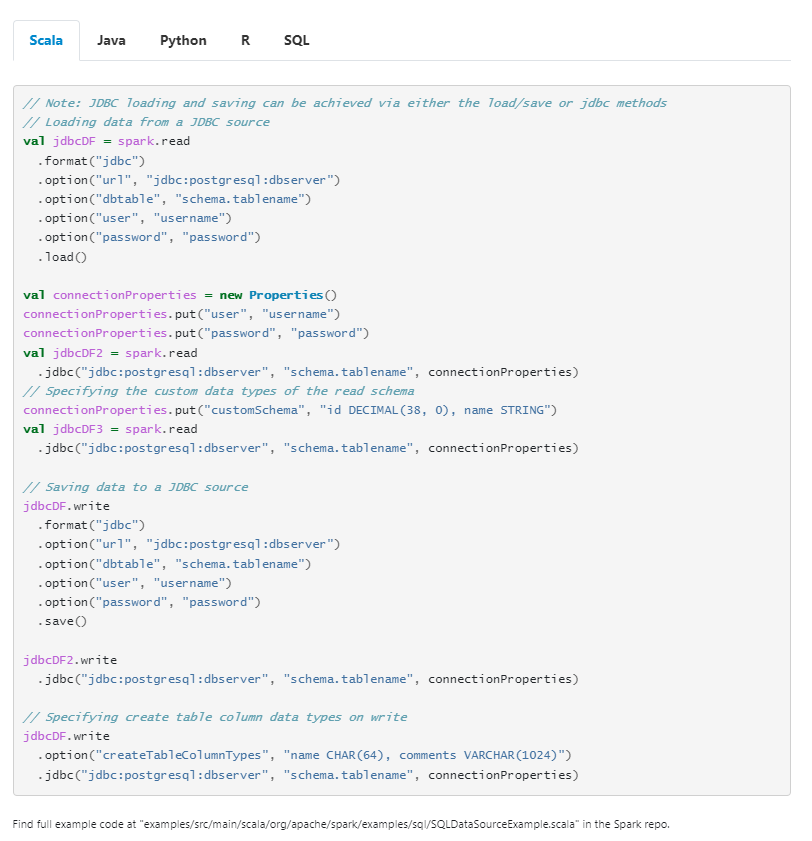
External Data Sources

JDBC and SQL Databases :

* Spark SQL includes a data source API that can read data from other database using JDBC
* It querying these data source and returns results as dataFrame (thus providing all of the benefits of Spark SQL, performance and ability to join other data source)
* To start
  + You should specify the JDBC driver for the JDBC data source and need to be on the spark classpath ($Spark\_HOME)  
    ./bin/spark-shell --driver-class-path $database.jar --jars $database.jar
  + Using data source API, tables from remote database can be loaded as a DataFrame or Spark SQL temporary view
  + JDBC connection proporeties can be specify through connection properties in the data source options

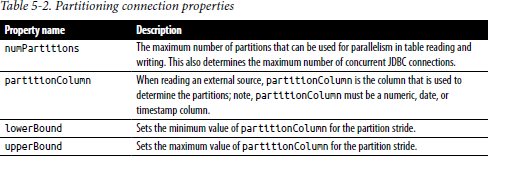






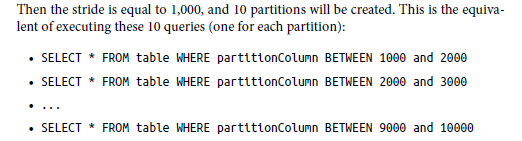
The importance of partitioning

* Important to partition data source when transferring large amounts data between spark SQL and JDBC external source
  + All of data is going through one driver connection, which can saturate and significantly slow down the performance of the extraction
  + JDBC properties for largeScale is recommended



* And example :
  + numPartitions: 10  
    lowerBound: 1000

upperBound: 10000

* + then the stride is equal to 1000 and 10 partitions will be created
  + is equivalent to   
    
  + for using numPartitions use a multiple of the number of spark wokers
    - Example: if there is 4 spark worker nodes => take 4 or 8 partitions
    - Be aware also how well your source system can handle read requests
      * For systems that have processing windows, you can maximize the number of concurrent requests to the source system
      * For systems lacking processing windows (OLTP system continuously processing data) the number of concurencies should be reduce to avoid saturation
  + Calculate LowerBound and upperBound based on the minimum and maximum partitionColumn actual values
    - Ex: {numPartitions: 10, lowerBound:1000, upperbound: 10000} but all the values are between 2000 and 4000, then only 2 f the 10 queries will doing a lot of work
    - In this scenario the best configuration will be {numPartitions:10, lowerBound:4000, upperBound;6000}
  + Choose a partitionColumn that can be uniformly distributed to avoid data skew => possible to generate a new one instead, choose a different pertitionColumn to more evenly distribute your partitions

PostgreSQL

* To conncect ot PostgreSQL database , build or download the JDBC jar from Maven and add it to the classpath JDBC jar from maven and add to classpath and then start spark shell and specifying that jar

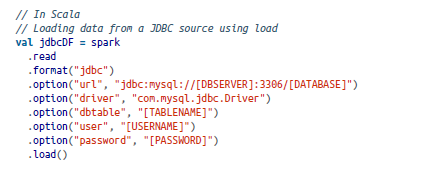
bin/spark-shell --jars postgresql-42.2.6.jar

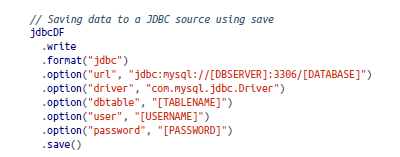




MySQL :

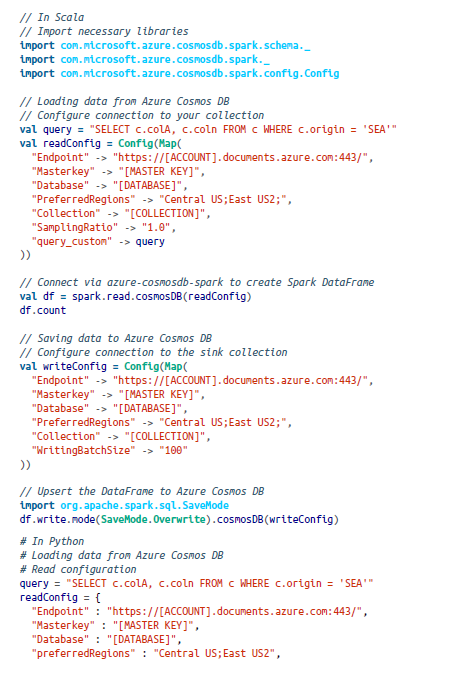
* To connect ot a MySQL database, build or download the JDBC jar from maven and add to classpath and then start spark shell and specifying that jar
* bin/spark-shell --jars mysql-connector-java\_8.0.16-bin.jar





Azure Cosmos DB

* To connect ot a MySQL database, build or download the JDBC jar from maven and add to classpath and then start spark shell and specifying that jar
* bin/spark-shell --jars azure-cosmosdb-spark\_2.4.0\_2.11-1.3.5-uber.jar
* also possible to pull connectore through Spark packages using its maven coordinates
* export PKG="com.microsoft.azure:azure-cosmosdb-spark\_2.4.0\_2.11:1.3.5"
* bin/spark-shell --packages $PKG





MS SQL Server

* To connect ot a MySQL database, build or download the JDBC jar from maven and add to classpath and then start spark shell and specifying that jar

bin/spark-shell --jars mssql-jdbc-7.2.2.jre8.jar



Other External Sources

* Spark can connect to these data source
  + Apache Cassandra
  + Snowflake
  + MongoDB

Higher-Order Functions in DataFrames and Spark SQL

* Complex data types are amalgamations of simple data types , to manipulate them there is two solution
  + Exploding the nested structure into individual rows, applying some function and then re creationg the nested structure
  + Bulding a user defined function
* These approaches have the benefit to allowing you to think og yje problem in tabular format
  + They typically involve using utility functions such as
    - Get\_json\_object
    - From\_json
    - To\_json
    - Explode
    - selectExpr()

Option 1: Explode and Collect

SELECT id, collect\_list(value + 1) as values

FROM (SELECT id, EXPLODE(values) as value  
 FROM table) x  
Group By id

* Collect\_list() retruns a list of object with duplicates
* GROUP BY statement requires shuffles operations, meaning the order of the recollected array isn’t necessarily the same as that of the original array
* As values could be any number of dimensions it can be very expensive (doing Group By)

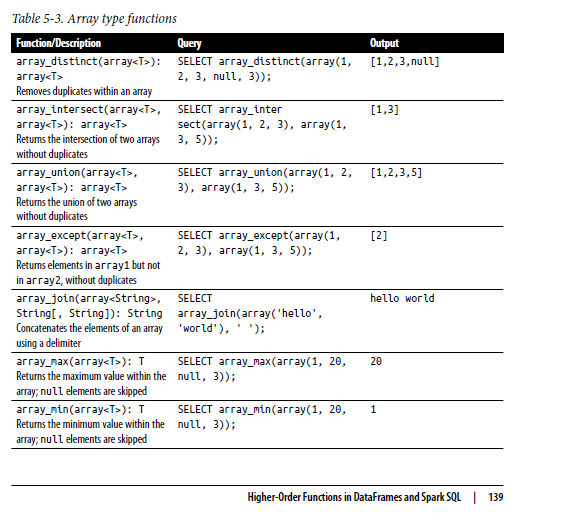
Option 2: User-Defined Function

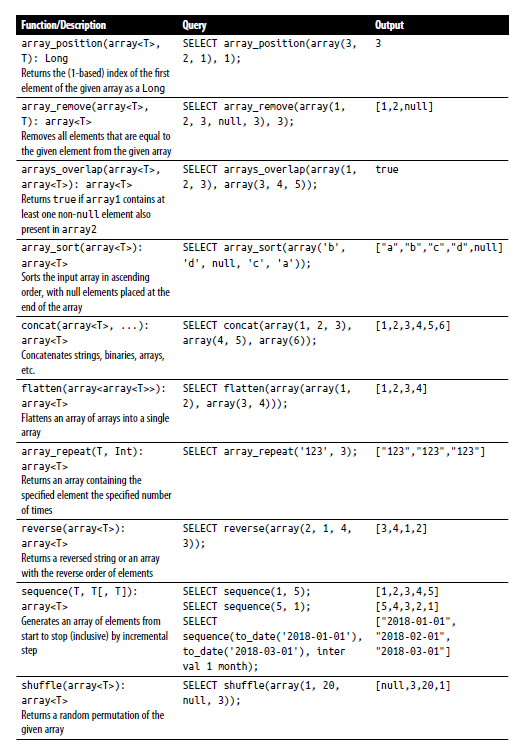
* To perfom same task as above another option could be to create UDF that uses map() to iterate through each element and perfom addition operation

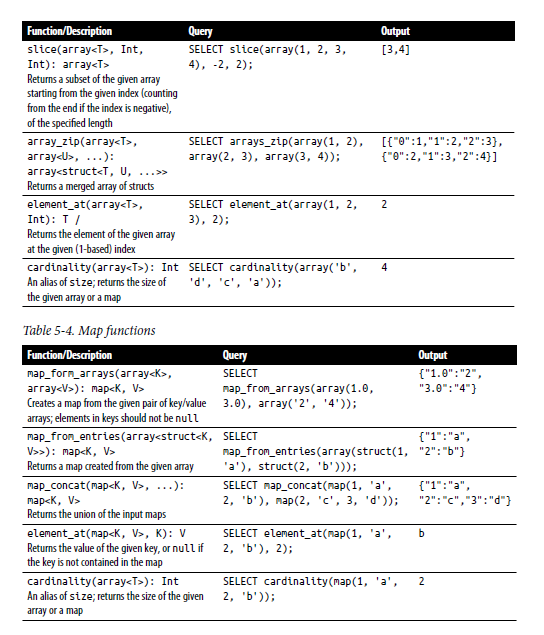
Spark.sql(“SELECT id, plusOneInt(values) as Values FROM tables”).show()

* While this is better ran using explode and collect\_list() as there won’t be any ordering issue , the serialization process itself may be expensive
* Collect\_list() could cause executors to experience out of memory issue for large data sets , whereas UDFs would alleviate these issues

Built-in Functions for Complex Data Types







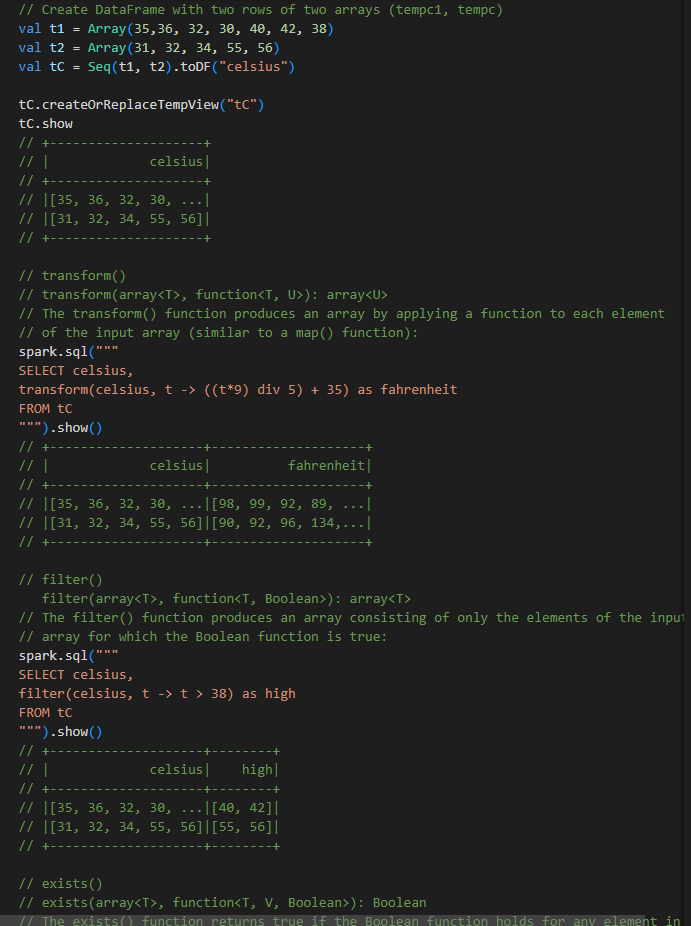
Higher-Order Functions

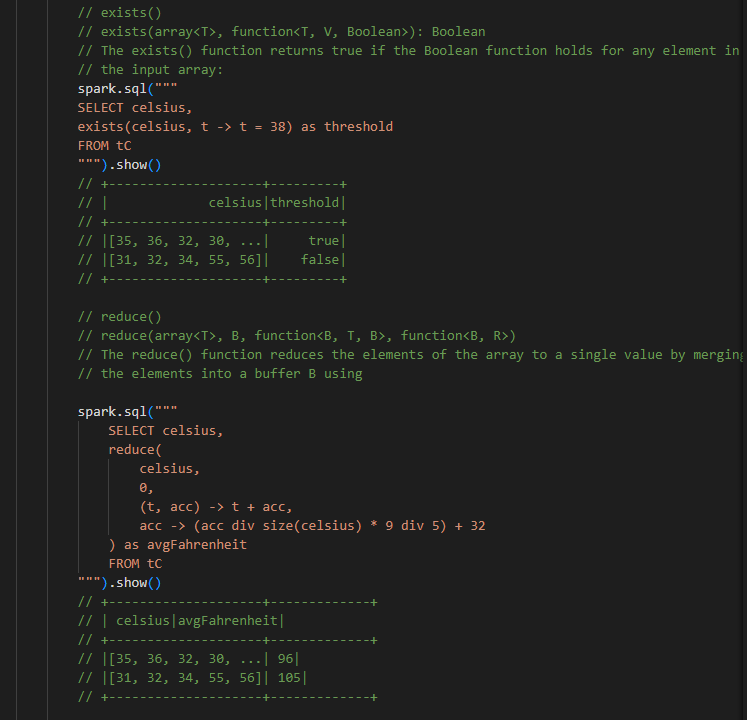
* There are Higher-order function that take anonymous lambda function as arguments an example :

*-- In SQL*

**transform**(values, value -> lambda expression)

* The Transform() function takes an array (values) and anonymous function (lambda expression) as input
* The function transparently creates a new array by applying thz anonymous function to each element and then assigning the result to the output array (similar to UDF approach)



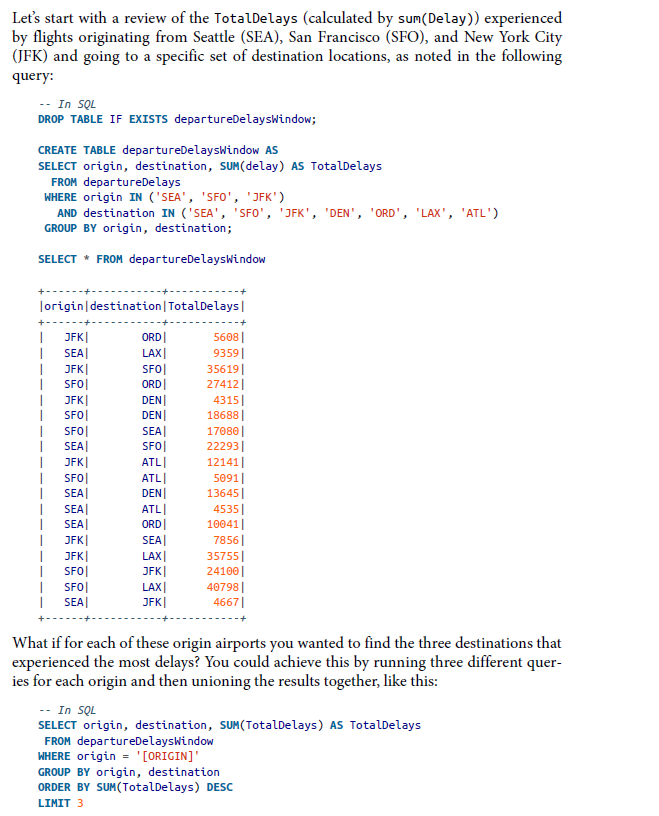


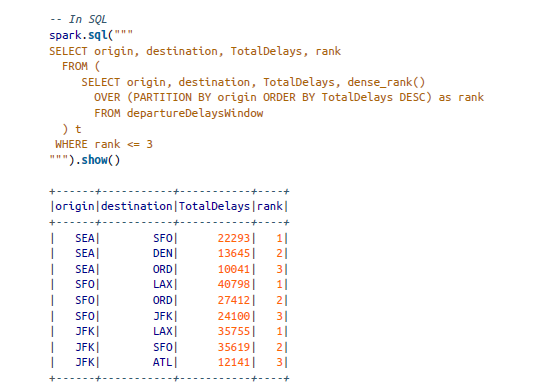
Common DataFrames and Spark SQL Operations

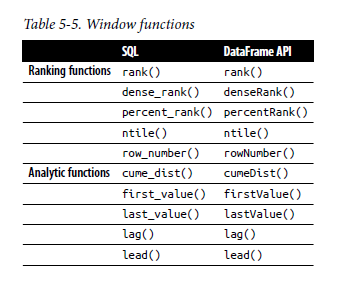
* DataFrame came with range of operation
  + Aggregate functions
  + Collection functions
  + Datetime functions
  + Math functions
  + Miscellaneous functions
  + Non-aggregate functions
  + Sorting functions
  + String functions
  + UDF functions
  + Window functions
* Union
  + Union should be apply on dataframe with the same schema
* Joins
  + Common DataFrame Operation is to Join 2 dataFrame tpgether
  + In Spark SQL join is an inner join with option being:
    - Inner
    - Cross
    - Outer
    - Full
    - Full outer
    - Left
    - Left\_outer
    - Rght
    - Right\_outer
    - Left\_semi
    - Left\_anti

Windowing

* A window function uses values from the rows in a window (a range of input rows) to return a set of values
* Typically in the form of another row
* With window functions it is possible to operate on a group of rows while still returning a single value for every input row

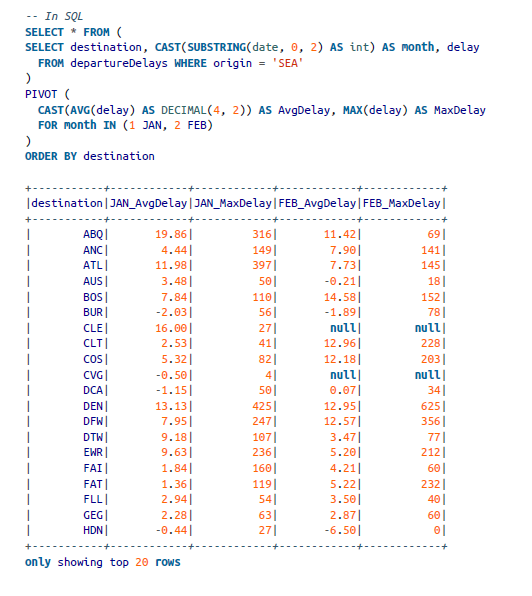






Modifications

* We can perform some modifications on dataframe
* While dataframes themselves are immutable, we can modify them through operations that create new



CHAPTER 6

Spark SQL and Datasets

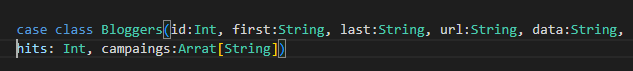
* Dataset is Strongly typed distributed collections (created, stored, serialized, deserialized)

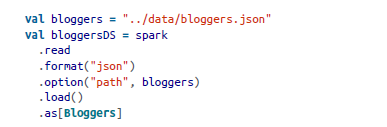
Single API for Java and Scala

* Dataset offer a unified and singular API for strongly typed objects (in java and scala)
* Python and R support only ubtyped DataFrame API
* Datasets are domain specific typed objects that can be operated on in parallel using functional programming or the DSL operators from DataFrame API

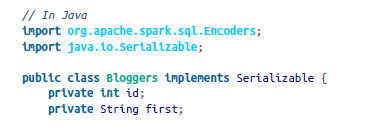
Scala Case Classes and JavaBeans for Datasets

* Spark has internal data type such as StringType, BinaryTypes, intergerType, BooleanType, MapType that use s to map seamlessly to the language specific fata types in scala and java during spark operatons
* Mapping is done through encoders
* To create DataSet[T] where T is the Typed object in scala we need case class that defines the object





* Each row in the resulting distributed data collection of type Bloggers
* Similary a Javabean class can be creadted





* Creating DataSets in Scala and JAVABean requires a bit of forethought as you have to know all the individual Column
* Dataset need that data type to be declare (case class , java been class)

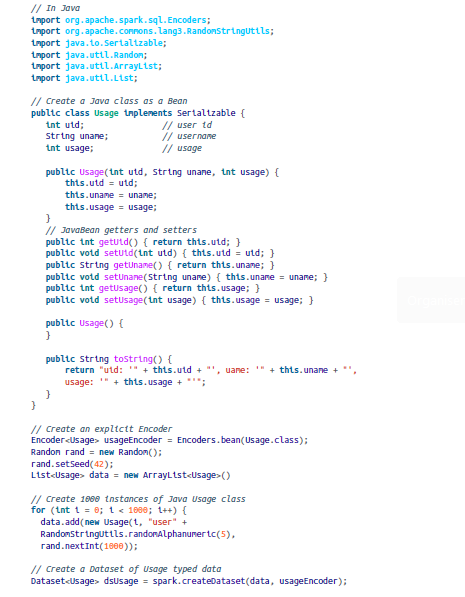
The names of the fields in the Scala case class or Java class definition match the order in the data source. The column names for each row in the data are automatically mapped to the corresponding names in the class and the types are automatically

preserved.

* Use Case class or java bean class to specify the schema

Working with Datasets

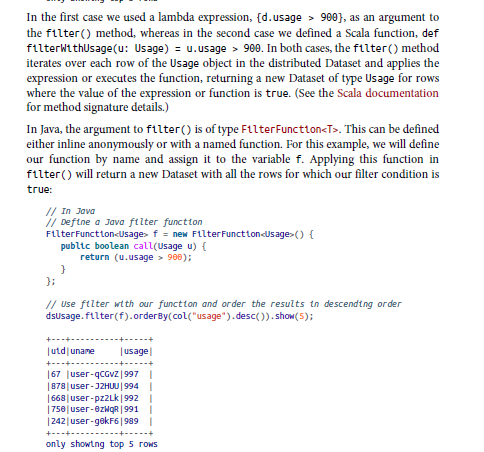
* Simple way to create dataets it use Sparksession instance



Transforming Sample Data

* Datasets are strongly typed collection of domain specific objects.
* These objects can be transformed in parallel using functional or relational operations  
  example transformations include map(), reduce(), filter(), select(), aggregate()
* As examples of high-order functions these methods can tale lambdas, closures or functions as arguments and return the results

Higher-order functions and functional programming



Memory Management for Datasets and DataFrames

* Spark is an intensive in memory distributed big data engine
* Need for use memory efficiently is crucial
* Memory evolved with spark evolution
  + Spark 1.0: used RDD based java object for memory storage, serialization and deserialization (exensive in terms of resources and slow)
    - Storage was allocated on the java heap so JVM garbage could impact large datasets
  + Spark 1.x introduced Project Tungsten:
    - One of its prominent features was a new internal row based format to lay out datasets and dataFrames in off heap memory
    - Using offset and pointer spark uses an efficient mechanism called encoders to serialize and deserialize between the JVM and its internal tungsten format
    - Allocating memory off-heaps menas that spark is less encumbered by GC
  + Spark 2.X introduced the second generation of Tungsten engine
    - Featuring whole stage code generation and vectorized column-based memory layout
    - Built on ideas and technique from modern compiler capitalized on modern CPU and cache architectures for fast parallel data access with Single instruction multiple data (SIMD approach)

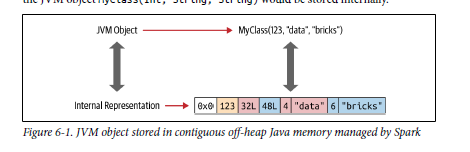
Dataset Encoders

* Encoders convert data in off-heap memory from spark internal Tungsten format to JVM java objects
* They serialize and deserialize Dataset object from Spark internal format to JVM object, including primitive data type
  + Ex: Encoder[T] will convert from spark internal Tungsten format to Dataset[T]
* Spark generate automatically encoders for primitive types (eg String, integer, long), Scala case clases and javaBeans
* Compared to Kryo serialization and deserialization, spark encoder are significantly faster.
* In java the encoder class is created explicitly

Encoder <UsageCost> usageCostEncoder = Encoder.bean(UsageCost.class)

* For scala spark autmotically generates the byte code for thes efficient converters

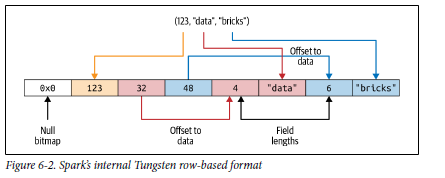
Spark’s Internal Format Versus Java Object Format

* Java object have large overheads – header info, hashcode, unicode info etc
  + Event simple java string such as “abcd” takes 48 bytes of storage instead of 4 bytes
* Instead of creating JVM-based object for dataset or dataframe
  + Spark allocates off-heap java memory to lay out their data and employs encoders to convert the data from in-memory representation to JVM object
  + 
  + When data is sotred in this continguous manner and accessible through pointer arithmetic and offsets, encoder can quickly serialize or deserialize that data

Serialization and Deserialization (SerDe)

* Is the process by which a typed object is encoded(serialized) into binary representation or format by the sender and decoded (deserialized) from binary format into respective data typed object by the receiver

EXemple: JVM object had to be shared among nodes in a spark cluster n the sender would serialize it into an array bytes and the receiver

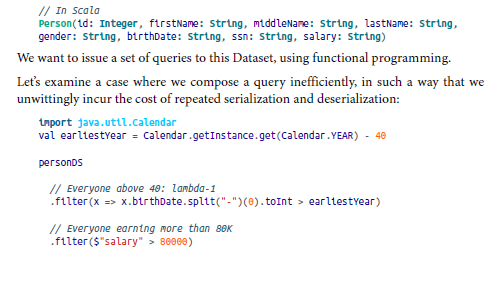
* The JVM has its own built in java serializer, but it’s inefiicient because (as we saw in previous section) the java objects created by the JVM in heap memory are bloated
* Spark internal Tungsten binary format stores objects off the java heap memory. And it’s compact so those objects occupy less space
* Encoder can quickly serialize by trabersing across the memory using simple pointer arithmetic with memory addresses and offsets
* On the receiving end, encoders can quickly desirialize the binary representation into spark internal representation . Encoder are not hindered by the JVM garbage collection pauses
* 

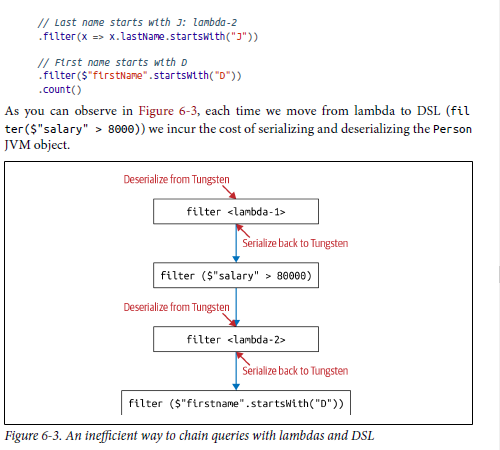
Costs of Using Datasets

* Use Datasets have a cost
  + When datasets are passed to higher-order functuins such a filter(), map(), that tales lambdas and functional arguments thereis cost associated with deserializing from spark internal tungsten format into the JVM object
  + Compared to encoder this cost is minor and tolerate, Howerer , over larger data srts and many queries this cost accrues and can affect performance

Strategies to Mitigate Costs

* One strategy to mitigate excessive serialization and deserialization is to sue DSL expression in the queries and avoid excessive use of lambdas as anonymous function as arguments to high order functions
* As lambdas are anonymous and opaque to the catalyst optimizer until runtime
* When use them it cannot efficiently discern what you are doing and thus cannot optimize your queries
* The second strategy is to chain your queries together in such way that serialization and deserialization is minimized





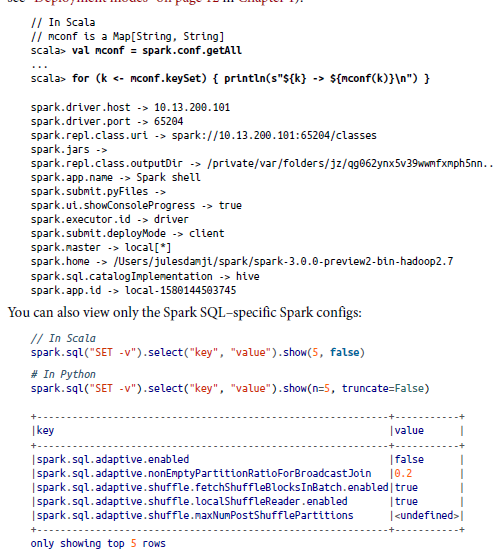
CHAPTER 7

Optimizing and Tuning Spark Applications

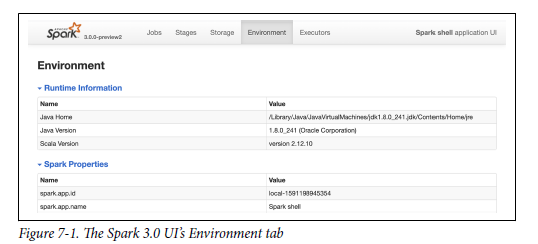
Optimizing and Tuning Spark for Efficiency

Viewing and Setting Apache Spark Configurations

* There a three ways to get and set spark properties
  + Configuration files
    - In the $SPARK\_HOME directory (where spark is installed) there are various config file
      * Conf/spark-default.conf.template
      * Conf/log4j.properties.template
      * Conf/spark-env.sh.template
      * Just need to change the default value and save it without .template
  + Specify the configuration directly in your spark application or on the command line when submitting the application with Spark-submit using –conf flag
    - 
    - Do it in spark his self
    - 
  + Through programmatic interface via the spark shell (as everything else in spark, APIs are the primary method of interaction( through spark session)



Alternative is to access conf via Spar UI



* To set or modify an existing configuration programmatically, first chech if the property is modifiable   
  SPARK.CONF.ISMODIFIABLE(“<config\_name>”): will return True or False
* 
* Among all the ways we set spark properties this is order of preference
  + Any values or flags defined in spark-defaults.conf will be read first
  + Follow by those supplied on the command line with spark-submit
  + And finally those set via sparkSession in the spark application
* All these properties will be merge with any duplicate properties reset in spark application taking precedence
* Value provided on the command line will supersede settings in the configuration file

Scaling Spark for Large Workloads

* Large Spark workloads are often batch jobs shcedulling at differenttime
* To avoid job failures due to resource starvation or gradual performance degradatation
* Use spark configuration which affect the spark component
  + Spark Driver
  + The Executor
  + Shuffle service running on the executor
* Spark dirver responsibility is to coordinate with the cluster manager to launch executors in a cluster and schedule spark tasks on them

Static versus dynamic resource allocation

* When specifying resources as command line arguments to spark-submit, the limit is capped
* If more resources are needed later as tasks queue up in the driver, spark can not accommodate or allocate extra resources
* One use case where this can be helpful is streaming, where data flow volume may be uneven
* Enabling resource allocation allow spark to achieve beter utilization of resources (freeubg executors when not in use and acquiring new ones when needed

As well as when working with large or varying workloads, dynamic allocation is also useful in a multitenant environment, where Spark may be deployed alongside other applications or services in YARN, Mesos, or Kubernetes. Be advised, however, that Spark’s shifting resource demands may impact other applications demanding

resources at the same time.

* To enabe dynamic allocation you can use these setting

spark.dynamicAllocation.enabled true

spark.dynamicAllocation.minExecutors 2

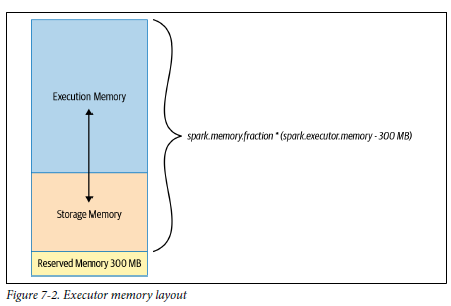
spark.dynamicAllocation.schedulerBacklogTimeout 1m

spark.dynamicAllocation.maxExecutors 20

spark.dynamicAllocation.executorIdleTimeout 2min

* By default spark.dynamicAllocation.enabled is set to false
* When enable spark driver will request that the cluster manager create 2 executor to start with , as a minimum
* As task backlog increases new executors will be requested eacg tume the backlog timeout (spark.dynamicAllocation.schedylerBacklogTimeout is excedd
* In this case, when a tasks have been scheduled for over 1min, the driver will request that a new executor be lauch it up to maximum of 20 spark.dynamicAllocation.maxExecutors
* Also if an executor finishes taks and is ide for 2 minutes ( spark.dynamicAllocation.IdleTimeout) spark driver will terminate it

spark.dynamicAllocation.maxExecutors

* To optimize spark we should also take into account how executor memory is laid out and used by spark, so that executors are not starved of memory or troubled by JVM garbage collection
* The amount of memory available to each executor is controlled by spark.executor.memory , this divided into the section :
  + Execution memory 60%(default)
  + Storage memory 40%
  + Reserved memory After allowing for 300 MB for reserved memory, to safeguard OOM errors
* The spark documentation advises that this will work for most cases, but you can adjust what fraction of spark.executor.memory you want
* When storage memory is not being used, spark can acquire it for use in execution memory for execution purposes and vice versa  
  
* Execution memory is used for spark shuffles, joins, sort and aggregation
  + Since different queties may require different amounts of memory, the fraction (spark.memory.fraction is 0.6 by default)
* Storage memory is primarily sued for caching user data structure and partitions derived from DataFRames
* During map and shuffle operations, spark writes to and reads from the local disk’s, spark read writes and reads from the local disk’s shuffles files, so there is heavy I/O activity.