

SUMMARY

Spark - Structured APIs

In this session, you were introduced to the concept of structured APIs in Apache Spark. You also learnt how these APIs are different from Resilient Distributed Datasets(RDDs). Then you used the Jupyter notebook to perform exploratory data analysis (EDA) using structured APIs. Further, you learnt about SparkSQL and Spark UI.

Introduction to Structured APIs

Let us begin by revising the limitations of RDDs and the need for structured APIs:

1. The **data** stored with the RDD abstraction is **unstructured**. While dealing with unstructured data, Spark recognises that there are parameters (or attributes) associated with each datapoint object, although it cannot read the object inside to get more details of the parameters.
2. RDD is a **low-level abstraction**. The code has very-low-level details about the execution of a job.

Structured APIs: The **data** stored in these abstractions is **structured**, that is, it is stored in the form of rows and columns. The **code** written in a structured API is **readable and intuitive**, just like writing a command using the Pandas library.

Let us understand the three structured APIs in Spark:

1. **DataFrames:** These are collections of data organised in a table format with rows and columns. They allow processing a large amount of structured data. One of the major differences between DataFrames and RDDs is that the data in DataFrames is organised in rows and columns, whereas in the case of RDDs, it is not. DataFrames do not have compile-time type safety.
2. **Datasets:** These are an extension of DataFrames, and they include the features of both DataFrames and RDDs. Datasets provide an object-oriented interface for processing data safely. Object-oriented interface refers to an interface where all entities are treated as objects, and in order to access the entities, one has to call the objects. Note that Datasets are available only in JVM-based languages - they are available in Scala and Java, but not in Python and R. Datasets have compile-time type safety.
3. **SQL Tables and Views (SparkSQL):** With SparkSQL, you can run SQL queries against views or tables organised into databases.

Introduction to Dataframes

The data in **DataFrames** is available in the form of columns and rows, with each column associated with a specific data type. This structure of columns with specific data types is called the schema of a DataFrame.

Some of the benefits of DataFrames are as follows:

- **High-Level API:** The code written in DataFrames is highly readable and easy to write. Since the code is at a high level, it becomes accessible to a lot of people other than specialists such as data engineers or data scientists.
- **Catalyst Optimiser:** Dataframes have the ability to optimise code internally. These optimisations are performed by the catalyst optimiser. It has the ability to rearrange the code to make it run faster without any difference in output.
- **Memory Management:** Transformations take place in the heap memory of the executors, which are essentially JVMs. On the other hand, DataFrames make use of the off-heap memory in addition to the heap memory on the JVMs. Hence, managing both heap memory and off-heap memory requires custom memory management.
- **Garbage Collection in JVM:** In languages like C/C++, the programmer is responsible for creating and destructing objects, although objects that are not useful and are usually neglected are not destroyed and remain in the memory. Over time, these objects keep on accumulating in memory, and sooner or later, the memory might become full with such garbage objects, resulting in out-of-memory errors. To deal with this, Java has a garbage collection (GC) program, which collects all such objects that are not in use and removes them from memory. One can say that the main objective of GC is to free up heap memory.

As the executor works on the job tasks, at some point, due to the increase in the amount of garbage objects, it has to run a 'full garbage collection' process. During this process, the machine slows down, as it has to scan the entire heap memory. So, the larger the JVM memory is, the more time that the machine will take to carry out this process. Note that GC runs on only JVM memory, which is heap memory.

- **Off-Heap Memory Storage:** An elegant solution to the problem above is to store data off the JVM heap but still in the RAM, and not on the disk. With this method, one can allocate a large amount of memory off-heap without GC slowing down the machine. If the data is stored off-heap, then the size of the JVM memory can be reduced and the GC process will not affect the performance as much since GC runs on only JVM memory, which is heap memory. Another advantage of off-heap memory is that the storage space required is much smaller compared with that for JVM memory, as Java objects in JVM require more storage space.

Datasets

A Dataset combines the advantages of RDDs, such as compile-time type safety, with the advantages of DataFrames, such as the catalyst optimiser and the ease of high-level programming. Let us revise the properties of a Dataset:

- **High-Level API:** If you compare a code written in RDDs with a code written in Datasets, then the code in Datasets would be readable and short.

- **Compile-Time Type Safety:** It is the ability of the Dataset abstraction to detect errors in the code at compile time itself.

	SQL	Dataframes	Datasets
Syntax errors	Runtime	Compile time	Compile time
Analysis errors	Runtime	Runtime	Compile time

- **Encoders:** The primary function of an encoder is the translation between the heap memory, that is, the JVM memory, and the off-heap memory. It achieves this by serialising and deserialising the data. Serialisation is the process of converting a JVM object to a byte stream. This conversion is necessary as the off-heap memory does not understand objects; it only understands byte streams. The data stored in the off-heap memory is serialised.

When a Spark job is run, the job query is first optimised by Spark. At runtime, the encoder takes this optimised query and converts it into byte instructions. In response to these instructions, only the required data is deserialised. This happens in the case of DataFrames as well.

- **Reduction in Memory Usage:** Datasets 'understand' the data as it is structured, and they can create efficient layouts to reduce memory usage while caching.

Dataframes Hands-On

You have already had hands-on with DataFrames. Now, let us quickly recall your learnings.

Following are the commands to initiate a Spark session:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Name of the app").getOrCreate()
```

The next step is to read in the data:

```
df = spark.read.csv("file name.csv", inferSchema = True, header = True)
```

You learnt about the different methods that are used to extract data row-wise and column-wise from a DataFrame. The commands are as follows:

- `.withColumn()` is used to create a new DataFrame with the specified columns.
- `.withColumnRenamed()` is used to rename columns.
- `.head()` is used to extract rows from the DataFrame.

The row object is used widely. Let us revise what a row object is and the different operations that can be performed on a row object.

Row Object: A row object is a collection of attributes and their values. Each datapoint (row) in a DataFrame is stored as a row object. The structure of the data can be gathered from the row object. Multiple similar row objects when

combined form a DataFrame.

- **filter()**

The filter method can be called directly on a DataFrame, and as an argument to the filter method, we provide a column object and specify the condition over the column. There are some other ways to apply the filter command. These are listed below.

The argument to the filter command can also be provided in the SQL syntax:

```
df.filter("column>500").show()
```

It can also be called using a column-type object:

```
df.filter(df['column']>500)
```

The filter command can have more than one operator, of which some are listed below:

Logical Operators	Symbols
and	&
or	
not	~
Equal to	==

- **collect()**

The collect() command is an action that allows you to fetch all the rows in a DataFrame. The output of the collect command is a list of row objects. Compared with the .show() command, which prints a DataFrame with only 20 rows, the collect command collects all the rows in a DataFrame.

- **groupBy()**

When the groupBy command is called alone, it just creates a groupBy object. And when the object is printed, the output is not a DataFrame that is segregated by the columns specified in the command; it is simply a groupBy object and its memory location.

- **Aggregation commands**

Merely the instructions to a group are not enough; the aggregation to be done on the group also needs to be specified. Some of the common aggregation functions include sum, max, min, etc.

Spark SQL

As you already know, Spark has the ability to run SQL queries, and the Spark module that is used to run SQL queries is called SparkSQL. SparkSQL allows us to run SQL- and HIVE-type queries on DataFrames. The power of SparkSQL is further enhanced by its ability to connect to a large number of standard databases via JDBC and ODBC, and it can collect data from a number of different file formats such as parquet, ORC, JSON, etc. A person habituated to writing SQL- and HIVE-type queries can easily switch to SparkSQL.

With the ability to connect to a wide variety of data sources and the simplicity of SQL-type queries, SparkSQL is quite a powerful tool and is accessible to a lot of people.

In order to run a query, we need a table, which can be created as follows:

```
createOrReplaceTempView('table_name')
```

This method converts a DataFrame to a temporary SQL table internally, making it ready to run SQL queries. Once we have the table ready, we can then run queries on the data directly. The data extracted from the queries is in the form of a DataFrame. All of the DataFrame functionalities that you learnt about earlier can be used to process the data further. This amalgamation of DataFrames and SQL queries imparts a lot of flexibility to the code.

Summary

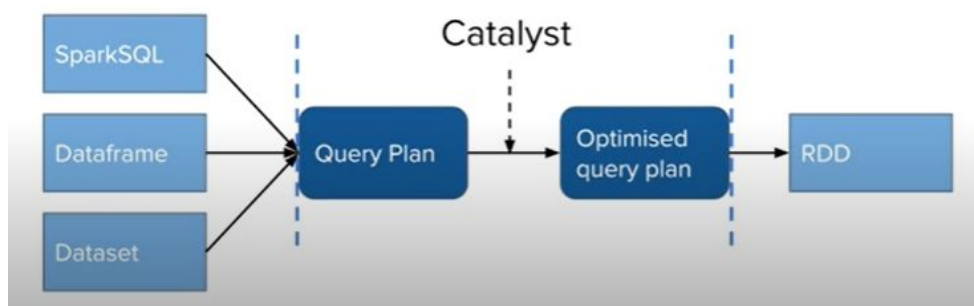
Optimisation in Structured APIs

In this session, you learnt about the catalyst optimiser and the working of the Spark UI.

Catalyst Optimiser

As you can recall, one of the major differences between RDDs and structured data abstractions is the internal optimiser, that is, the Catalyst Optimiser. The catalyst optimiser forms the core of SparkSQL. Its main function is to optimise a high-level code in such a way that it takes the least amount of time to execute.

You can compare DAGs in RDDs to the catalyst optimiser in the case of structured APIs. Now, let us take a look at the different steps of a structured API:



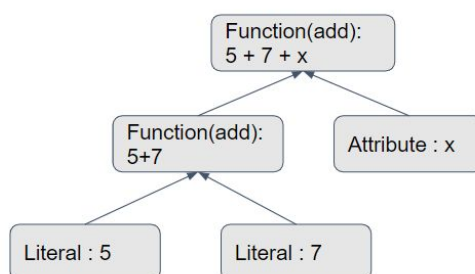
- Query:

Any query that you write gets optimised by the catalyst optimiser in Spark. The way a query is written affects the logical plan.

TREE:

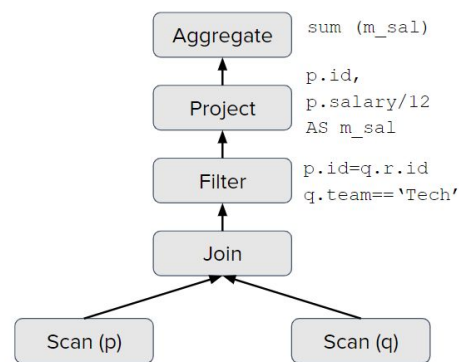
- The structure representing the logical flow of a query is called a tree.
- It is a fundamental data type of the optimiser.
- Each of the nodes in a tree is an object.

The diagram below shows a basic version of a tree:



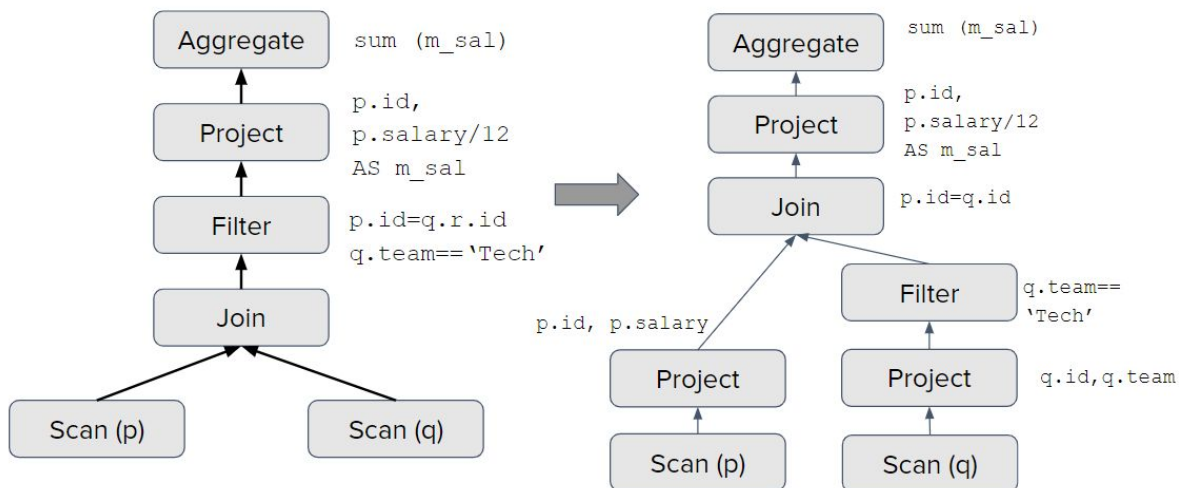
● Logical Plan

The code written is read and arranged to form a sequence of operations in a bottom-up manner thus forming a logical sense of the query. This arrangement may or may not be the most optimised one. The following diagram shows the sequence of the steps involved in a logical plan:



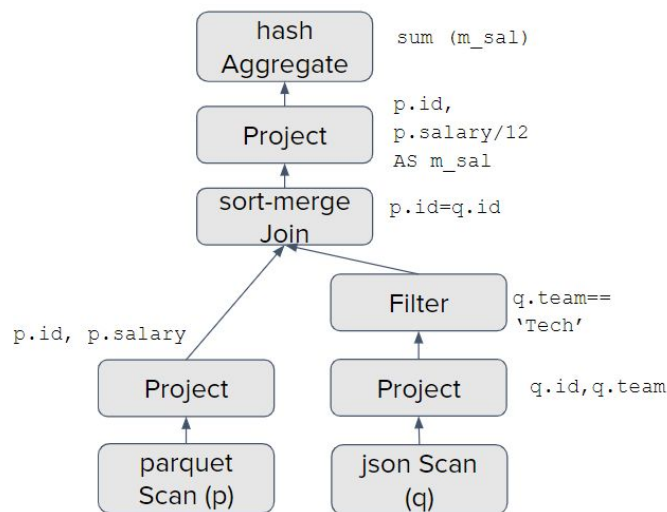
● Optimised Plan

Optimisation is a rule-based approach. Once the logical plan is created, it is used to form an optimised logical plan. The diagram below depicts an optimised plan:



● Physical Plan

After the logical plan is created, the process of creating a physical plan begins. A physical plan is a high-level approach to the logical plan. The diagram below shows the physical plan:



● Cost Model

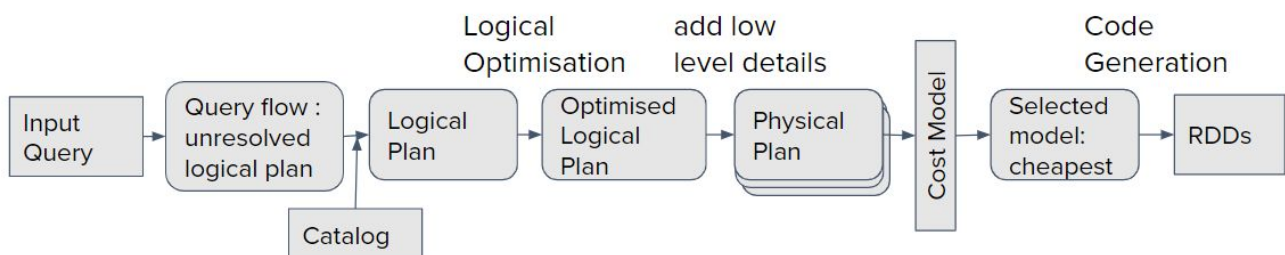
The Cost Model is the only step of the catalyst optimiser that is not rule-based. It is a cost-based step. You can enable the cost model as follows:

```
spark.conf.set("spark.sql.cbo.enabled","true")
```

Multiple physical plans are formed from an optimised logical plan. The physical plans are then fed into the cost model, which performs a cost-based analysis on them. It estimates the computational effort required for each operation based on the number of rows in each table and the size of the output, and then selects the most optimised plan.

● Architecture of Catalyst Optimiser

We have already revised the catalyst optimiser. Now, let's take a look at the architecture of the catalyst Optimiser as a whole:



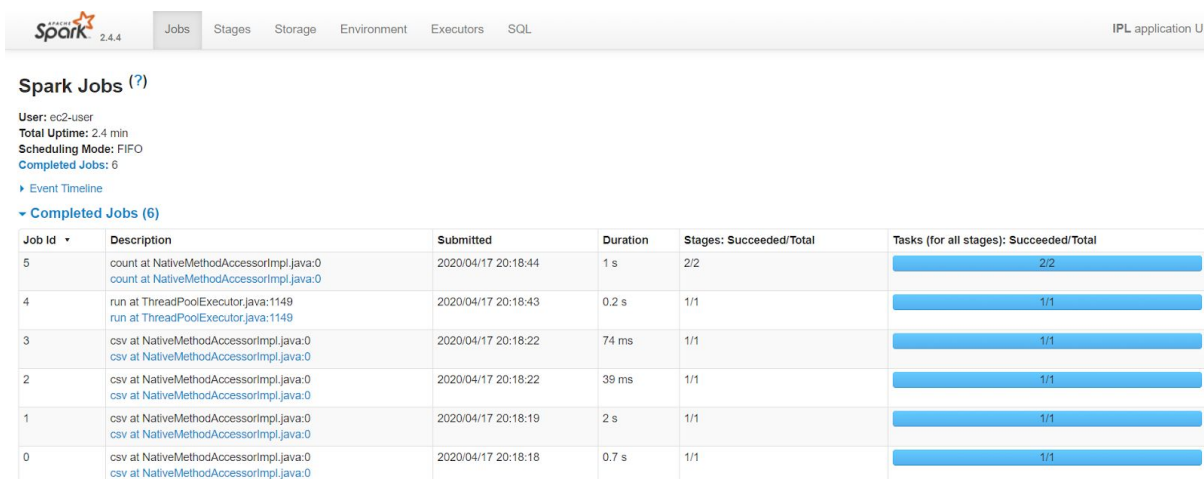
In the diagram above, you can see that the optimisation plan is broadly divided into four parts:

1. **Analysis:** This step involves analysing the query and forming a logical plan using various relation names from the catalog, and mapping attributes where needed.

- Logical Optimisation:** As mentioned earlier, this is purely a rule-based approach and, hence, it forms a rule-based optimised logical plan.
- Physical Planning:** In this step, the logical plan is picked up and used to form one or more physical plans, which are then sent to the cost model, where a cost-based analysis is carried out.
- Code Generation:** As the name suggests, this step focuses on the creation of the RDD-level code. The bytecodes that are generated by the optimiser run on the JVM, and are faster and a bit different from the Java bytecodes that are generated by RDDs. Another advantage here is that bytecodes can also be executed across different machines directly, thereby making the Spark cluster implementation even more efficient.

Introduction to SparkUI

As you have learnt previously, the Spark UI is a web interface that lets you visualise the resource consumption of a Spark cluster. It is an important tool for improving the performance of a Spark cluster. This is how the Spark UI looks like:



The screenshot shows the Spark UI interface with the 'Jobs' tab selected. It displays a table of completed jobs. The table has columns for Job Id, Description, Submitted, Duration, Stages, and Tasks. The tasks are represented by blue progress bars.

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
5	count at NativeMethodAccessorImpl.java:0 count at NativeMethodAccessorImpl.java:0	2020/04/17 20:18:44	1 s	2/2	2/2
4	run at ThreadPoolExecutor.java:1149 run at ThreadPoolExecutor.java:1149	2020/04/17 20:18:43	0.2 s	1/1	1/1
3	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2020/04/17 20:18:22	74 ms	1/1	1/1
2	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2020/04/17 20:18:22	39 ms	1/1	1/1
1	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2020/04/17 20:18:19	2 s	1/1	1/1
0	csv at NativeMethodAccessorImpl.java:0 csv at NativeMethodAccessorImpl.java:0	2020/04/17 20:18:18	0.7 s	1/1	1/1

This SparkSQL web UI displays information about the following:

- Tasks scheduled:**
Whenever an action is executed, a job entry is created on the job page. The job entry details, the duration of a job, its stages, etc. To get more details about a particular job, you need to click on the corresponding description link.
- Amount of memory in use:**
Here, you can see the stages of a job and the memory that is used to execute it, along with the DAG representation.
- Catalyst trees:**
You can reach here by going to the SQL tab then clicking on a job. This diagram depicts the optimisation and physical planning for a particular job.
- Information about executors:**
It shows the performance of each individual executor when there are more than one executors working.

Summary

Structured APIs - Usage

In this session, you learnt about the different sources from which data can be extracted, and established the difference between RDDs and structured APIs.

Sources of Data

You learnt about the different sources from which data can be extracted in Spark structured APIs. Let us quickly revise these sources. Apart from imparting flexibility through different APIs, Spark can also perform the following functions:

- Gathering data from different sources and
- Reading/writing files of different formats.

In case you want to read data from different files, the syntax is as follows:

```
spark.read.load("filename.fileformat", format = "fileformat", inferSchema = True, header = True)
```

But you do not need to mention the file format for Parquet files as they are the default files in Spark. You can write in the files using the following command:

```
df.write.save("filename.fileformat", format = "fileformat")
```

Similarly, you can use the following command for running SQL queries on unread files:

```
spark.sql(" select * from parquet.'<address and name for the file>' ")
```

Now, let us quickly revise the different sources for gathering data:

1. **Databases:** Java Database Connectivity, or JDBC, is a built-in Spark functionality that allows users to connect to other external databases such as the Microsoft SQL server.
2. **HIVE Tables:** Spark can also read and write data from HIVE tables. However, connecting to HIVE directly is not enough; you need to specify all the dependencies of HIVE in the classpath. Also, since you will be working on a cluster, the dependencies should be present in all the worker nodes.
3. **Table Partitioning:** This is a popular technique that is used to divide the data into smaller, manageable chunks. For example, consider the database of the customers on an e-commerce website. As each and every activity that is performed by the customers on the platform is recorded, the volume of data becomes huge.

Recall the features of both RDDs and structured APIs, and try to understand which API should be used under what circumstances:

- You should use RDD abstractions in the following situations
 1. **Low-level APIs:** When you need high control over the method of executing jobs. The MapReduce-style commands in RDDs allow users to pass detailed instructions about the execution of a job.
 2. **Schema is unavailable:** When the schema of the data is either not available or is not relevant to the user.
 3. **Data is unstructured:** When the data used in the analysis is unstructured, e.g., log files, text data, etc.
 4. **Optimisation is not a priority:** When optimisation is not important. In other words, the aforementioned points are more important than the time taken to execute a code, that is, the speed of execution of code.
- You should use structured APIs in the following scenarios
 1. **High-level APIs:** When ease of execution of code is an important factor. The code written in a high-level API can be read easily. The instructions provided in the code are related to the objectives that it needs to achieve.
 2. **Structured data:** When the data sources are structured/semi-structured, e.g., csv, json, DB, parquet, etc.
 3. **Availability of libraries:** High-level APIs have many pre-existing libraries for machine learning. This reduces the coding effort.
 4. **Optimisation:** When the code needs to be executed quickly. Structured APIs have an in-built optimiser capability called the catalyst optimiser, which allows users to do this.

Disclaimer: All content and material on the upGrad website is copyrighted material, either belonging to upGrad or its bonafide contributors and is purely for the dissemination of education. You are permitted to access, print and download extracts from this site purely for your own education only and on the following basis:

- You can download this document from the website for self-use only.
- Any copy of this document, in part or full, saved to disk or to any other storage medium, may only be used for subsequent, self-viewing purposes or to print an individual extract or copy for non-commercial personal use only.
- Any further dissemination, distribution, reproduction, copying of the content of the document herein or the uploading thereof on other websites or use of the content for any other commercial/unauthorised purposes in any way which could infringe the intellectual property rights of upGrad or its contributors, is strictly prohibited.
- No graphics, images or photographs from any accompanying text in this document will be used separately for unauthorised purposes.
- No material in this document will be modified, adapted or altered in any way.
- No part of this document or upGrad content may be reproduced or stored in any other website or included in any public or private electronic retrieval system or service without upGrad's prior written permission.

- Any right not expressly granted in these terms are reserved.